全新打造的Qt爱好者社区强力支持！
短期即可入门及提高的优秀Qt系列书籍！

Qt应用编程系列丛书

霍亚飞 编著
吴 迪 白建平 董世明 审校

# Qt及Qt Quick开发实战精解

- 全新，基于最新的Qt及Qt Creator编写，包含Qt Quick！
- 经典，基于经典的Qt网络博客编写，可无限更新！
- 综合，对众多知识点进行综合应用，实例经典实用！
- 系统，与《Qt Creator快速入门》配套，理论结合实际！

北京航空航天大学出版社
BEIHANG UNIVERSITY PRESS

Qt 应用编程系列丛书

# Qt 及 Qt Quick 开发实战精解

霍亚飞 编著

吴 迪 白建平 董世明 审校

北京航空航天大学出版社

## 内 容 简 介

本书主要讲解了5个Qt综合应用程序的开发过程和Qt Quick的相关内容。本书内容主要包括两部分：第一部分是多文档编辑器、方块游戏、音乐播放器、数据管理系统、局域网聊天工具这5个实用的Qt实例的详细讲解；第二部分是Qt Quick技术的全面介绍。

本书的内容全面、实用，讲解通俗易懂，适合有一定Qt基础并且想学习Qt综合实例开发或者想学习Qt Quick技术的读者。对于没有Qt基础的读者，可以先学习《Qt Creator快速入门》一书。

**图书在版编目(CIP)数据**

Qt及Qt Quick开发实战精解 / 霍亚飞编著. -- 北京
：北京航空航天大学出版社，2012.5
ISBN 978-7-5124-0781-7

Ⅰ. ①Q… Ⅱ. ①霍… Ⅲ. ①移动终端—应用程序—
程序设计 Ⅳ. ①TN929.53

中国版本图书馆CIP数据核字(2012)第063827号



**Qt及Qt Quick开发实战精解**

霍亚飞 编著

吴 迪 白建平 董世明 审校

责任编辑 董立娟

*

北京航空航天大学出版社出版发行

北京市海淀区学院路37号(邮编100191) http://www.buaapress.com.cn

发行部电话：(010)82317024 传真：(010)82328026

读者信箱：emsbook@gmail.com 邮购电话：(010)82316936

涿州市新华印刷有限公司印装 各地书店经销

*

开本：710×1 000 1/16 印张：17 字数：372千字

2012年5月第1版 2012年5月第1次印刷 印数：4 000册

ISBN 978-7-5124-0781-7 定价：36.00元

# 丛书前言

本系列书籍的原型是网络上的一系列博客教程。2009 年，当作者开始使用推出不久的 Qt 开发环境 Qt Creator 时，由于该软件还不成熟，所以经常出现一些问题，但是在网络上却几乎找不到任何的相关内容可以参考。于是作者便将自己的学习经验和心得进行整理，以 yafeilinux 为网名在博客中推出了一系列 Qt 和 Qt Creator 相关教程。这些教程浅显易懂，使用的描述语言很自然，而且图文并茂，每一个知识点都设计了一个可以运行的示例程序。由于这些特点都很符合读者的需求，所以这个系列教程得到了众多网友的肯定和称赞，这也给了作者很大的信心继续写下去。到 2010 年，作者接到北航出版社的邀请，希望可以将系列教程整理成书。由于当时想写成一本与众不同的书籍，要与市面上已经出版的书籍产生差异，所以一直等待着 Qt 和 Qt Creator 全新版本的发布，直到现在才完成本套书籍的编写。

在这一年里，作者创建了 yafeilinux.com 网站，希望可以有更多的人一起参与到 Qt 的普及工作中。在该网站创建后，作者不仅将系列教程扩展到了 Qt 基础、2D 绘图、数据库操作和网络通信等几个部分，而且还推出了多个专题教程和开源软件。其中，《Qt 串口通信专题教程》在网络上广为流传，几乎成为学习 Qt 串口编程的必备教程；而开源软件中的“音乐播放器”，在推出不到一个月的时间里就有了上万次的下载量。现在网站的访问量已经超过百万，同时在作者的邮箱中，一年里收到了两千多封请教和交流的邮件，而 QQ 交流群更是建一个满一个。这些都让作者看到了 Qt 爱好者的热情和 Qt 发展的良好趋势，也让作者下定决心要写出一本好书。2011 年 3 月，Qt 4.7.2 和 Qt Creator 2.1.0 正式发布，该版本中 Qt Creator 第一次正式使用了中文界面，也是第一次正式支持 Qt Quick 编程。作者感觉是时候开始正式写作了，有了这一年来和众多网友的交流，以及作者在 Qt 各个应用方面的编程积累，已经有了足够的信心来写出一部经典作品。

直至今日书已成型，10 个月的写作包含了太多的心酸与喜悦，最终呈现给读者的是包括了《Qt Creator 快速入门》和《Qt 及 Qt Quick 开发实战精解》两部作品的一个系书籍。之所以同时推出两部作品，而没有将其合成一部书籍，是为了给不同学习阶段的读者提供灵活的选择。尽管本个系书籍已经包含了 Qt 中最常用的大部分内容，不过 Qt 实在太庞大了，所以还是有很多内容没有写进去，这些会在 yafeilinux.com 网站中得到补充。一本书的厚度是有限的，但是网络中的内容却可以是无限的，

网上的系列教程，作者还会一直写下去的。

## Qt 简介

Qt 是一个跨平台应用程序和 UI 开发框架。使用 Qt 只需一次性开发应用程序，无须重新编写源代码，便可跨不同桌面和嵌入式操作系统部署这些应用程序。Qt Software 的前身为创始于 1994 年的 Trolltech(奇趣科技)，Trolltech 于 2008 年 6 月被 Nokia 收购，加速了其跨平台开发战略。Qt Creator 是全新的跨平台 Qt IDE，可单独使用，也可与 Qt 库和开发工具组成一套完整的 SDK。其中包括：高级C++代码编辑器、项目和生成管理工具、集成的上下文相关的帮助系统、图形化调试器、代码管理和浏览工具。Qt Quick 是在 Qt 4.7 中被引进的一种高级用户界面技术，开发人员和设计人员可用它协同创建动画触摸式用户界面和应用程序。

## 系列丛书特色

本系列丛书作为全面介绍 Qt、Qt Creator 和 Qt Quick 的入门级教材，也是市面上第一套详细介绍 Qt Creator 和 Qt Quick 的教材。与其他相关书籍最大的不同之处还在于，本套书籍是基于网络博客教程的。综合来说，本套书籍主要具有以下特色：

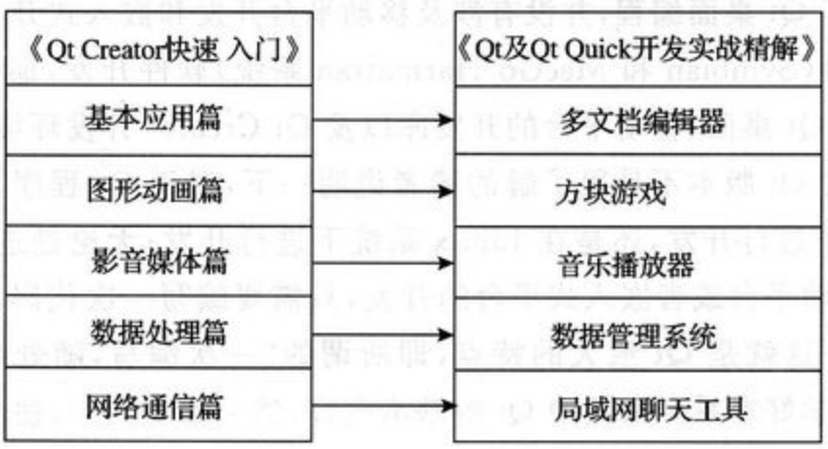
- 最新。基于最新的 Qt 4.7.2 和 Qt Creator 2.1.0 版本进行编写，在该版本中首次正式集成了 Qt Quick。
- 最全。书中的内容包含了 Qt 基础、图形动画、多媒体、数据库、网络通信、WebKit 以及 Qt Quick 等所有基本的应用内容。
- 无限更新。本套书籍对应的网络教程是无限更新的，而且读者可以通过邮件、QQ 群或者论坛和作者零距离接触。
- 按应用分类。本套书籍中将同一应用的相关章节放在一起作为一个篇章，例如图形动画篇中讲解了所有与绘图和动画相关的知识，这样组织更有利于读者在程序开发时进行选择性参考学习。
- 知识点讲解与大型实例设计分离。本套书籍将知识的讲解和综合实例的设计分为了两部分。这样读者便不会因为要在很长的代码中学习某个知识点而一头雾水，但每一个知识点都设计了一个很简单的示例程序，让读者可以实际地看到该知识点的应用效果；而设计大型实例时，将主要的精力放在程序框架和功能实现上，不再细讲其中的知识点，这样更有利于读者学习设计综合性程序。
- 程序设计分步实现。本套书籍中不是像其他书籍那样将一个完整示例程序的所有代码全部列出并进行讲解，而是分步来设计该程序，从创建项目开始，一

个功能模块一个功能模块的分步添加，这样可以让读者更好地了解程序的设计过程。

➢ 关键字提示。本套书籍的编写主要基于 Qt 参考文档，而且所讲解的知识点也只是 Qt 参考文档中的部分内容，读者在学习时一定要多参考 Qt 帮助文档。在本套书籍讲解的所有知识点和示例程序中，都很明显地标出了其在 Qt 帮助中对应的关键字，这样可以让读者对书中的内容有迹可循。

## 丛书结构

本套书籍一共分为两本：《Qt Creator 快速入门》和《Qt 及 Qt Quick 开发实战精解》。《Qt Creator 快速入门》中主要进行知识点的详细讲解，根据应用的不同分为了 5 个篇章：基本应用、图形动画、影音媒体、数据处理和网络通信；在《Qt 及 Qt Quick 开发实战精解》中主要讲解了 5 个综合实例以及 Qt Quick 的内容，其中的 5 个实例程序分别对应了《Qt Creator 快速入门》的 5 个篇章，也就是说每一个应用实例都是对应篇章知识点的综合应用。本套书籍的结构如下图所示：




## 使用本系列丛书籍

对于没有任何 Qt 编程经验的读者，建议先学习《Qt Creator 快速入门》。该书对 Qt 基本内容以及 Qt Creator 开发环境进行了详细讲解，而且每一个知识点都设计了一个可以独立运行的示例程序，非常适合初学者入门使用。对于有一定 Qt 编程经验，并且想学习 Qt 综合实例编写的读者，推荐使用《Qt 及 Qt Quick 开发实战精解》。在该书中对于不同的应用领域分别设计了一个实用的应用程序，而且每一个实例都应用了众多的知识点，非常适合没有实际开发经验的读者。如果读者想学习全新的 Qt Quick 技术，那么《Qt 及 Qt Quick 开发实战精解》也是很好的选择，该书的 Qt Quick 部分详细介绍了 Qt Quick 的方方面面。

当然，本套书籍是一个相互联系的整体，建议读者最好可以在学习《Qt Creator

快速入门》的同时学习《Qt 及 Qt Quick 开发实战精解》，每学完一篇内容就去学习对应的实例程序，这样可以达到更好的效果。

在学习过程中要多动手，尽量按照步骤编写代码，只有在遇到自己无法解决的问题时再去参考本系列书籍提供的源代码。每当学习一个知识点时，书中都会给出 Qt 帮助中的关键字，建议参考 Qt 的帮助文档，看英文原文是怎么描述的。不要害怕去看那些英文文档，因为不可能在网上找到所有文档的中文翻译，而且有些翻译也偏离了原意，所以最终还是要自己去看英文的参考文档的。学会看参考文档是入门 Qt 编程的重要一步，不要怕自己英文不好，其实跟自己的英文水平关系不大，只要坚持，掌握了一些英文术语和关键词以后，阅读英文文档是不成问题的。

## Qt 版本说明

本系列书籍是基于 Windows 下的 Qt 4.7.2 和 Qt Creator 2.1.0 版本的，这两个版本是在本系列书籍开始编写时的最新版本。为了避免读者使用不同的操作系统而产生不必要的问题，本系列书籍采用了最常用的 Windows XP 系统。并且本系列书籍中只讲解了 Qt 桌面编程，并没有涉及移动平台开发和嵌入式开发的内容，如果要进行移动平台(Symbian 和 MeeGo Harmattan 系统)软件开发，那么可以下载 Qt SDK，它集成了 Qt 桌面、移动平台的开发库以及 Qt Creator 开发环境。

这里要向对 Qt 版本不是很了解的读者说明一下，对于 Qt 程序开发，无论是在 Windows 系统下进行开发，还是在 Linux 系统下进行开发；无论是进行桌面程序开发，还是进行移动平台或者嵌入式平台的开发，只需要编写一次代码，然后分别进行编译就可以了。这就是 Qt 最大的特点，即所谓的“一次编写，随处编译”。也就是说，读者只需要学好本系列书籍中 Qt 的基本内容，然后编写代码，使用 Qt 不同的版本进行编译即可。

在学习本系列书籍时，推荐读者使用指定的 Qt 和 Qt Creator 版本，因为对于初学者来说，任何微小的差异都可能导致错误的理解。当然，也可以使用其他版本，在作者的网站上有不同平台的各个版本的使用教程。

## 致　谢

感谢北京航空航天大学出版社的相关人员，是他们的邀请和支持，才让作者坚定信心要写一本与众不同的经典作品。

感谢那些关注和爱好 Qt 的朋友们，是他们的支持和帮助，才让作者一步一步走下来。其中一些朋友还参与了本系列书籍的审校和代码审核工作，具体的审校分工是：解放军装甲兵工程学院的吴迪(wd007)完成了初审、Qt 中文论坛管理员白建平(XChinux)完成了二审、上海大学计算机工程与科学学院的董世明完成了终审；参加

代码审核的朋友有：程梁（豆子）、刘柏桑（紫侠）、么贺文（mehewen）、胡峰（古月行云）、周慧宗、杨凡（云帆）和黄金金等，他们分别完成了本套书籍代码在 Windwos XP 系统、Windows 7 系统和 Ubuntu Linux 系统下的代码审核工作。是众多朋友的认真工作，才使得本系列丛书籍可以较早地出版。

由于作者技术水平有限，经验也很欠缺，再加上 Qt 的内容繁多，并且没有统一的中文术语参考，所以书中难免有各种错误理解和代码设计缺陷，恳请读者批评指正。读者可以到作者的网站 www. yafeilinux. com 下载本套书籍的源码，查看与本套书籍对应的不断更新的系列教程。也可以到 Qt 爱好者社区（www. qter. org）在本系列丛书的专版进行讨论交流。

霍亚飞

2012 年 2 月于北京

# 前　言

本书主要讲解了5个Qt综合应用程序的开发过程和Qt Quick的相关内容，适合有一定Qt基础并且想学习Qt综合实例开发或者想学习Qt Quick技术的读者。本书中的实例部分没有涉及太多知识点的讲解，而是将重点放在了知识的应用上。与本书对应的《Qt Creator快速入门》，该书主要讲解Qt的知识点，本书中的几个实例就是对该书讲解的知识点的综合应用。对于没有Qt基础，或者在学习本书实例遇到困难时，建议先学习《Qt Creator快速入门》中相应的基础知识。

本书内容共包含了6章，分为两个部分：综合实例和Qt Quick。综合实例部分共包含5章，分别对应《Qt Creator快速入门》的每一个篇章设计了一个综合实例程序：

- 第1章　多文档编辑器：对应基本应用篇。该实例实现了一个可以同时处理多个子文档的编辑器软件，主要包含了菜单设计、主窗口应用和各功能模块间相互约束等内容。
- 第2章　方块游戏：对应图形动画篇。该实例实现了一个经典的俄罗斯方块游戏，主要包含了游戏逻辑设计、图形视图应用、动画设置和添加背景音乐等内容。
- 第3章　音乐播放器：对应影音媒体篇。该实例实现了一个可以显示桌面歌词的音乐播放器软件，主要包含了Phonon框架应用、文件解析、2D绘图和系统托盘图标等内容。
- 第4章　数据管理系统：对应数据处理篇。该实例实现了一个使用数据库管理数据并使用饼状图显示数据的管理系统，主要包含了数据库、XML和自定义视图等内容。
- 第5章　局域网聊天工具：对应网络通信篇。该实例实现了一个可以在局域网中发送信息和文件的通信工具软件，主要包含了UDP编程、TCP编程和富文本处理等内容。

Qt Quick部分只包含了第6章，主要讲解了Qt最新引入的Qt Quick技术，虽然Qt Quick编程与《Qt Creator快速入门》中讲述的Qt C++编程有着千丝万缕的联系，但是它是以一种全新的技术推出的，其中还包含了一门全新的语言QML。

霍亚飞

2012年2月于北京

# 目　录

## 第一部分　综合实例

## 第二部分 Qt Quick

# 第一部分

# 综 合 实 例

# 第 1 章　多文档编辑器

这一章的例子是对《Qt Creator 快速入门》基础应用篇各章节知识的综合应用，也是一个规范的实例程序。之所以说其规范，是因为在这个程序中，我们对菜单什么时候可用/什么时候不可用、关闭程序时应该先保存已修改且尚未保存的文件等细节都做了严格的约束。而一个真正实用的应用程序，也就应该如此。

本章应用了基础篇的众多知识点，但这里只是讲解程序流程与框架，不会涉及太多知识细节的讲解，所以希望读者可以学完基础应用篇后再来学习本章。这个实例主要是对主窗口部件的应用，所以可以学完《Qt Creator 快速入门》的前 5 章再来学习本章，这样可以达到更好的效果。该实例是基于 Qt 中的 MDI Example 示例程序的，它在 Main Windows 分类下。这个程序就是以 QMainWindow 类为主窗口，以 QMdiArea 类为多文档区域，以 QTextEdit 类为子窗口部件，从而实现了一个多文档编辑器的应用。最终的运行效果如图 1-1 所示。

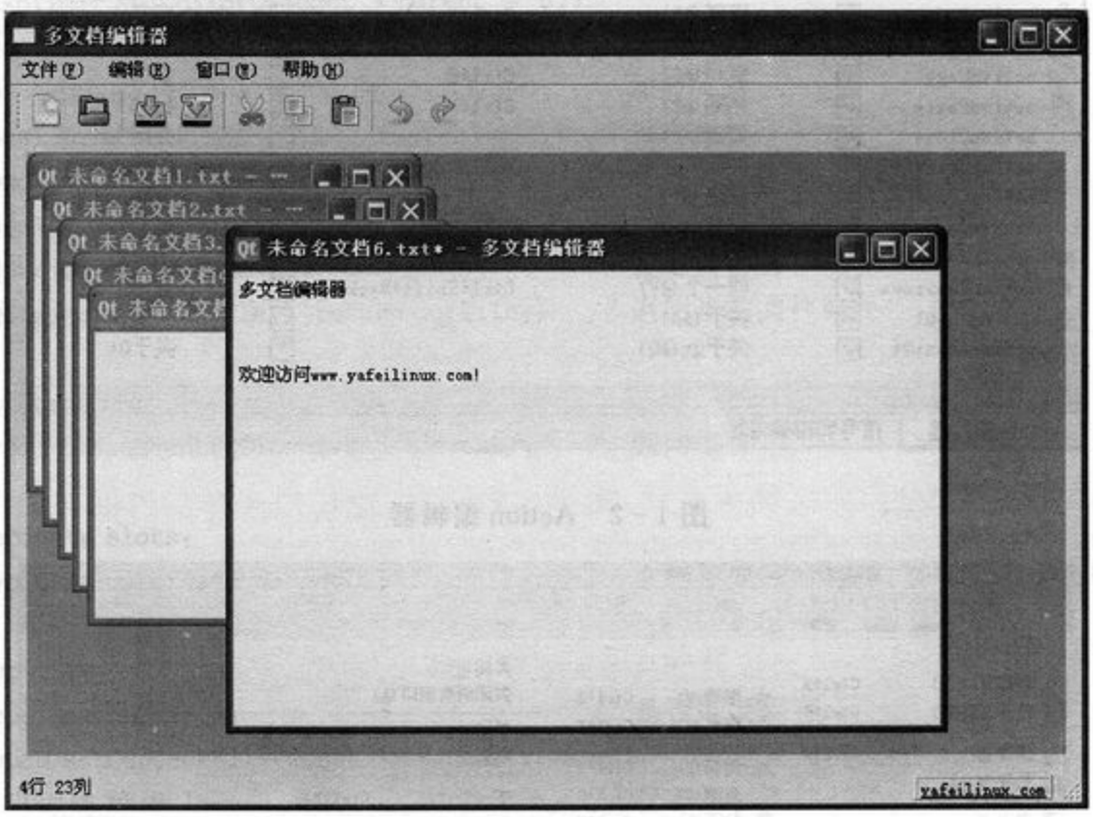



图 1-1　多文档编辑器界面

## 1.1 界面设计

首先进行界面的设计，这里主要是对主窗口菜单栏和工具栏的设计。打开 Qt Creator，创建新的项目。（项目源码路径：src\1\1-1\myMdi）新建 Qt Gui 应用，项目名称 myMdi，类名默认为 MainWindow，基类默认为 QMainWindow 都不做改动。完成后双击 mainwindow.ui 文件进入设计模式，然后添加各个菜单，所有的菜单动作如图 1-2 所示，最终的菜单栏和工具栏如图 1-3 所示。设计菜单时，如果将来触发这个菜单会弹出一个对话框进行详细设置，那么就在这个菜单文本后面添加“...”号，例如这里的“打开文件”菜单和“另存为”菜单。这里还要注意，添加动作时，一定要使动作名称和这里的 Action 编辑器中所使用的名称保持一致，因为在后面的程序中还要用到它们。

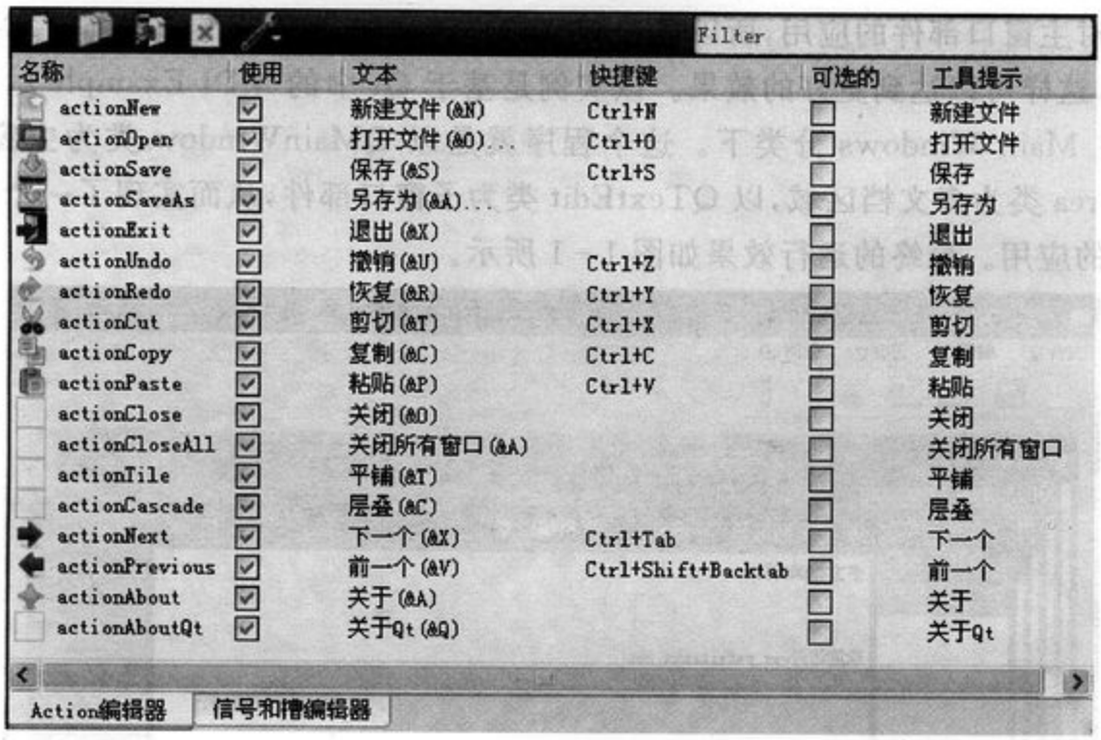



图 1-2 Action 编辑器

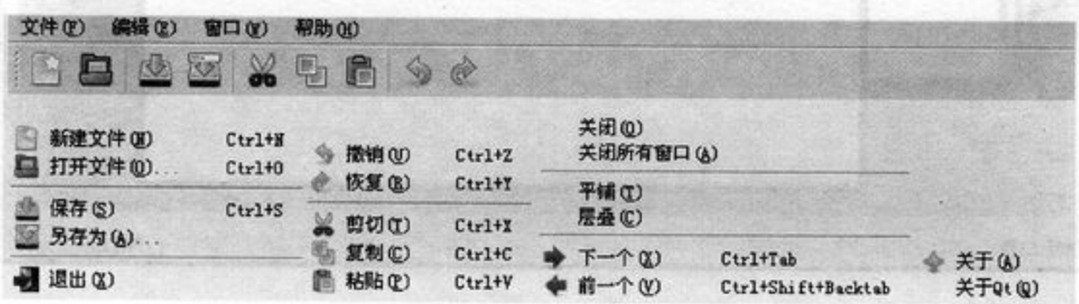



图 1-3 菜单栏与工具栏

设计完菜单栏与工具栏后，向主窗口中心区域拖入一个 MdiArea 部件，并单击主窗口界面，按下 Ctrl+G 快捷键，使其处于栅格布局之中。可以看一下对象列表窗

口，确保 MdiArea 部件的 objectName 是 mdiArea，而文件菜单、编辑菜单、窗口菜单和帮助菜单的 objectName 分别是 menuF、menuE、menuW 和 menuH；如果不是，需要在属性栏中更改，因为后面的程序中要用到。

## 1.2　创建子窗口类

为了实现多文档操作，需要向 QMdiArea 中添加子窗口，而为了可以更好地操作子窗口，必须子类化子窗口的中心部件。因为这里子窗口的中心部件使用了 QTextEdit 类，所以要实现自己的类，它必须继承自 QTextEdit，然后在其中添加我们的功能函数。

（项目源码路径：src\1\1 - 2\myMdi）往项目中添加新文件，模板选择“C++ 类”，类名为 MdiChild，基类为 QTextEdit，类型信息选择“继承自 QWidget”。完成后在 mdichild.h 文件中添加代码：

```
class MdiChild : public QTextEdit
{
  Q_OBJECT
  public:
  explicit MdiChild(QWidget * parent = 0);
  void newFile();                                  // 新建操作
  bool loadFile(const QString &fileName);          // 加载文件
  bool save();                                     // 保存操作
  bool saveAs();                                   // 另存为操作
  bool saveFile(const QString &fileName);          // 保存文件
  QString userFriendlyCurrentFile();               // 提取文件名
  QString currentFile(){return curFile;}           // 返回当前文件路径

  protected:
  void closeEvent(QCloseEvent * event);            // 关闭事件

  private slots:
  void documentWasModified();                      // 文档被更改时，窗口显示更改状态标志

  private:
  bool maybeSave();                                // 是否需要保存
  void setCurrentFile(const QString &fileName);    // 设置当前文件
  QString curFile;                                 // 保存当前文件路径
  bool isUntitled;      // 作为当前文件是否被保存到硬盘上的标志
};
```

这里在头文件中声明了 11 个函数，定义了两个变量。其中，currentFile() 函数返回当前的文件路径，只有一行代码，就直接在这里定义了。所以真正需要设计的只

有 10 个函数，还有 curFile 与 isUntitled 两个变量，分别用于保存当前文件的路径和作为文件是否被保存过的标志。因为对于所有的应用程序，只有涉及新建、保存和关闭等操作时，都是使用的这些函数进行设置的，它们是一个整体，所以这里要将它们同时罗列出来。这些函数主要完成了下面几个操作：

**1. 新建文件操作 newFile()**

- 设置窗口编号；
- 设置文件未被保存过“isUntitled = true;”；
- 保存文件路径，给 curFile 赋初值；
- 设置子窗口标题；
- 关联文档内容改变信号到显示文档更改状态标志槽 documentWasModified()。

**2. 加载文件操作 loadFile()**

- 打开指定的文件，并读取文件内容到编辑器；
- 设置当前文件 setCurrentFile()，该函数可以获取文件路径，完成文件和窗口状态的设置；
- 关联文档内容改变信号到显示文档更改状态标志槽 documentWasModified()。

**3. 保存操作 save()**

- 如果文件没有被保存过(用 isUntitled 判断)，执行另存为操作 saveAs()；
- 否则直接保存文件 saveFile()，该函数先打开指定文件，然后将编辑器的内容写入该文件，最后设置当前文件 setCurrentFile()。

**4. 另存为操作 saveAs()**

- 从文件对话框获取文件路径；
- 如果路径不为空，则保存文件 saveFile()。

**5. 关闭操作 closeEvent()**

- 如果 maybeSave()函数返回为真，则关闭窗口。maybeSave()函数判断文档是否被更改过，如果被更改过，则弹出对话框，让用户选择是否保存更改，或者取消关闭操作。如果用户选择保存更改，则返回保存操作 save()的结果，如果选择取消，则返回 false。否则，直接返回 true。
- 如果 maybeSave()函数返回为假，则忽略该事件。

下面到 mdichild.cpp 文件中添加代码。这里先列出所有需要添加的头文件，当然，实际写程序时，是用到哪个类才去添加那个类的头文件的。

```
#include <QFile>
#include <QTextStream>
#include <QMessageBox>
#include <QFileInfo>
#include <QApplication>
```

```
#include <QFileDialog>
#include <QCloseEvent>
#include <QPushButton>
```

然后在 MdiChild 类的构造函数中添加两行代码：

```
// 设置在子窗口关闭时销毁这个类的对象
setAttribute(Qt::WA_DeleteOnClose);

// 初始 isUntitled 为 true
isUntitled = true;
```

第一行代码会在后面讲解关闭操作时提到，而第二行代码只是对变量的初始化。下面开始介绍 5 个操作的函数实现。

首先是新建文件操作：

```
void MdiChild::newFile()
{
  // 设置窗口编号，因为编号一直被保存，所以需要使用静态变量
  static int sequenceNumber = 1;

  // 新建的文档没有被保存过
  isUntitled = true;

  // 将当前文件命名为未命名文档加编号，编号先使用再加 1
  curFile = tr("未命名文档 %1.txt").arg(sequenceNumber++);

  // 设置窗口标题，使用[*]可以在文档被更改后在文件名称后显示"*"号
  setWindowTitle(curFile + "[*]" + tr(" - 多文档编辑器"));

  // 文档更改时发射 contentsChanged()信号，执行 documentWasModified()槽
  connect(document(), SIGNAL(contentsChanged()),
      this, SLOT(documentWasModified()));
}
```

这里在设置窗口标题时添加了"[*]"字符，它可以保证编辑器内容被更改后，在相应位置显示"*"号。下面是 documentWasModified()槽的定义：

```
void MdiChild::documentWasModified()
{
  // 根据文档的 isModified()函数的返回值，判断编辑器内容是否被更改了
  // 如果被更改了，就要在设置了[*]号的地方显示"*"号，这里会在窗口标题中显示
  setWindowModified(document()->isModified());
}
```

编辑器内容是否被更改，可以使用 QTextDocument 类的 isModified()函数获知，这里使用了 QTextEdit 类的 document()函数来获取它的 QTextDocument 类对

象。然后使用 setWindowModified()函数设置窗口的更改状态标志“*”,如果参数为 true,那么就会在标题中的设置了“[*]”号的地方显示“*”号,表示该文件已经被修改。

下面是加载文件操作:

```
bool MdiChild::loadFile(const QString &fileName)
{
  // 新建 QFile 对象
  QFile file(fileName);
  // 只读方式打开文件,出错则提示,并返回 false
  if (! file.open(QFile::ReadOnly | QFile::Text)) {
    QMessageBox::warning(this, tr("多文档编辑器"),
                         tr("无法读取文件 %1:\n%2.")
                         .arg(fileName).arg(file.errorString()));
    return false;
  }

  // 新建文本流对象
  QTextStream in(&file);
  // 设置鼠标状态为等待状态
  QApplication::setOverrideCursor(Qt::WaitCursor);
  // 读取文件的全部文本内容,并添加到编辑器中
  setPlainText(in.readAll());
  // 恢复鼠标状态
  QApplication::restoreOverrideCursor();
  // 设置当前文件
  setCurrentFile(fileName);
  connect(document(), SIGNAL(contentsChanged()),
    this, SLOT(documentWasModified()));
  return true;
}
```

在加载文件操作中使用了 QFile 类对象,它可以打开指定的文件,并且与 QTextStream 类配合使用,方便进行文件的读取与写入操作,关于这部分内容,读者可以参考《Qt Creator 快速入门》的第 15 章。建立 QMessageBox 时使用了 tr()函数,其中的“1%”和“2%”分别可以被后面的 arg()中的 fileName 和 file.errorString()代替,这样就可以在字符串中使用变量了,而“\n”表示换行的内容在《Qt Creator 快速入门》的第 7 章中涉及。在读取文件完成后还调用了 setCurrentFile()函数,下面是它的定义:

```
void MdiChild::setCurrentFile(const QString &fileName)
{
  // canonicalFilePath()可以除去路径中的符号链接,“.”和“..”等符号
  curFile = QFileInfo(fileName).canonicalFilePath();
```

```
    // 文件已经被保存过了
    isUntitled = false;

    // 文档没有被更改过
    document()->setModified(false);

    // 窗口不显示被更改标志
    setWindowModified(false);

    // 设置窗口标题,userFriendlyCurrentFile()返回文件名
    setWindowTitle(userFriendlyCurrentFile() + "[*]");
}
```

这个函数只是将加载文件的路径保存到 curFile 中，然后进行了一些状态的设置，在设置标题时使用了 userFriendlyCurrentFile()函数，下面是它的定义：

```
QString MdiChild::userFriendlyCurrentFile()
{
    // 从文件路径中提取文件名
    return QFileInfo(curFile).fileName();
}
```

该函数是从文件路径中提取出文件名，这样使标题显得更加清晰和友好，这就是这个函数名的含义。这里使用了 QFileInfo 类，也在《Qt Creator 快速入门》的第 15 章中讲到。

下面是保存操作：

```
bool MdiChild::save()
{
    // 如果文件未被保存过,则执行另存为操作,否则直接保存文件
    if (isUntitled) {
        return saveAs();
    } else {
        return saveFile(curFile);
    }
}
```

这里先使用 isUntitled 判断文件是否被保存过，如果没有，则要先进行另存为操作，如果已经保存过了，那么直接写入文件就可以了。下面是另存为操作函数的定义：

```
bool MdiChild::saveAs()
{
    QString fileName = QFileDialog::getSaveFileName(this, tr("另存为"),curFile);
    // 获取文件路径,如果为空,则返回 false,否则保存文件
```

```
    if (fileName.isEmpty())
      return false;
    return saveFile(fileName);
}
```

另存为操作中就是先使用文件对话框获取文件路径，如果路径不为空，则进行文件的写入，它由 saveFile()函数执行：

```
bool MdiChild::saveFile(const QString &fileName)
{
  QFile file(fileName);
  if (! file.open(QFile::WriteOnly | QFile::Text)) {
    QMessageBox::warning(this, tr("多文档编辑器"),
                         tr("无法写入文件 %1:\n%2.")
                         .arg(fileName).arg(file.errorString()));
    return false;
  }
  QTextStream out(&file);
  QApplication::setOverrideCursor(Qt::WaitCursor);
  // 以纯文本文件写入
  out << toPlainText();
  QApplication::restoreOverrideCursor();
  setCurrentFile(fileName);
  return true;
}
```

这个函数进行文件的写入操作，可以看到，它与 loadFile()函数是相对应的两个操作。

最后是关闭操作，在关闭事件中执行：

```
void MdiChild::closeEvent(QCloseEvent * event)
{
  // 如果 maybeSave()函数返回 true,则关闭窗口,否则忽略该事件
  if (maybeSave()) {
    event->accept();
  } else {
    event->ignore();
  }
}
```

事件在《Qt Creator 快速入门》的第 6 章中讲到。这里的关闭事件可以在窗口被关闭或者调用 close()函数时执行。如果执行了 accept()函数，那么窗口将会关闭，这里需要说明，其实窗口关闭，默认只是将窗口隐藏起来了，并没有将它销毁掉。但是因为在前面的构造函数中使用了“setAttribute(Qt::WA_DeleteOnClose);”这行代码，所以当关闭窗口时，它会被销毁掉。而如果执行了 ignore()函数，那么这个事

件会被忽略掉，什么都不做。然后就是那个 maybeSave()函数，它在窗口关闭时判断文件是否需要保存，并让用户进行选择：

```
bool MdiChild::maybeSave()
{
  // 如果文档被更改过
  if (document()->isModified()) {
    QMessageBox box;
    box.setWindowTitle(tr("多文档编辑器"));
    box.setText(tr("是否保存对“%1”的更改?")
                .arg(userFriendlyCurrentFile()));
    box.setIcon(QMessageBox::Warning);
    // 添加按钮，QMessageBox::YesRole 可以表明这个按钮的行为
    QPushButton *yesBtn = box.addButton(tr("是(&Y)"),QMessageBox::YesRole);
    box.addButton(tr("否(&N)"),QMessageBox::NoRole);
    QPushButton *cancelBtn = box.addButton(tr("取消"),
                                           QMessageBox::RejectRole);
    // 弹出对话框，让用户选择是否保存修改，或者取消关闭操作
    box.exec();
    // 如果用户选择是，则返回保存操作的结果；如果选择取消，则返回 false
    if (box.clickedButton() == yesBtn)
      return save();
    else if (box.clickedButton() == cancelBtn)
      return false;
  }
  // 如果文档没有更改过，则直接返回 true
  return true;
}
```

这个函数中进行文档是否被更改过的判断，如果文档被更改了，就弹出警告框，这里为了使警告框中的按钮可以显示中文，所以自定义了按钮。

这就是整个操作的过程，读者可以好好分析一下各个函数及其联系，因为以后写相似的应用程序时，这几个操作都是必须的，这里搞明白了，以后直接使用即可。下面对这个类进行简单的测试。

进入设计模式，在 Action 编辑器中“新建文件”动作上右击，转到它的触发信号 triggered()的槽，并更改如下：

```
void MainWindow::on_actionNew_triggered()
{
  // 创建 MdiChild
  MdiChild *child = new MdiChild;
  // 多文档区域添加子窗口
  ui->mdiArea->addSubWindow(child);
```

```
    // 新建文件
    child->newFile();
    // 显示子窗口
    child->show();
}
```

在这里新建了子窗口，并且以 MdiChild 类对象为中心部件，然后新建文件并且显示出来。这里大家要在 mainwindow.cpp 文件中添加 #include "mdichild.h"头文件，因为程序中使用了中文，所以还要在 main.cpp 文件中使用 setCodecForTr()函数进行设置。最后运行程序，然后按下 Ctrl+N 新建文件，并更改内容，然后进行关闭，看一下程序的运行效果。

## 1.3 实现菜单的功能

上一节建立了自己子窗口的中心部件 MdiChild 类，它继承自 QTextEdit 类。下面便可以使用这个类，来完成主窗口上的各个菜单的功能。

### 1.3.1 更新菜单状态与新建文件操作

(项目源码路径：src\1\1-3\myMdi)首先更新菜单状态，使一些菜单在开始时处于不可用状态。然后再更改新建文件的操作。

**1. 更新菜单状态**

在 mainwindow.h 文件中添加类 MdiChild 的前置声明："class MdiChild;"，然后添加私有槽 private slots：

```
void updateMenus();                    // 更新菜单
```

再添加 private 私有变量和函数：

```
QAction * actionSeparator;     // 间隔器
MdiChild * activeMdiChild();   // 活动窗口
```

这里的 actionSeparator 动作用于创建一个间隔器，将来在窗口菜单中显示子窗口列表时，可以用它与前面的菜单动作分隔开。下面在 mainwindow.cpp 文件中添加代码。

先添加头文件 #include <QMdiSubWindow>，然后在 MainWindow 类的构造函数中添加如下代码：

```
// 创建间隔器动作并在其中设置间隔器
actionSeparator = new QAction(this);
actionSeparator->setSeparator(true);
```

```
    // 更新菜单
    updateMenus();

    // 当有活动窗口时更新菜单
    connect(ui->mdiArea, SIGNAL(subWindowActivated(QMdiSubWindow *)), this, SLOT(updateMenus()));
```

这里初始化了 actionSeparator 动作，然后执行更新菜单函数，并关联多文档区域的活动子窗口信号到更新菜单槽上，每当更换活动子窗口时，都会更新菜单状态。

```
    void MainWindow::updateMenus()
    {
      // 根据是否有活动窗口来设置各个动作是否可用
      bool hasMdiChild = (activeMdiChild() != 0);
      ui->actionSave->setEnabled(hasMdiChild);
      ui->actionSaveAs->setEnabled(hasMdiChild);
      ui->actionPaste->setEnabled(hasMdiChild);
      ui->actionClose->setEnabled(hasMdiChild);
      ui->actionCloseAll->setEnabled(hasMdiChild);
      ui->actionTile->setEnabled(hasMdiChild);
      ui->actionCascade->setEnabled(hasMdiChild);
      ui->actionNext->setEnabled(hasMdiChild);
      ui->actionPrevious->setEnabled(hasMdiChild);

      // 设置间隔器是否显示
      actionSeparator->setVisible(hasMdiChild);

      // 有活动窗口且有被选择的文本，剪切复制才可用
      bool hasSelection = (activeMdiChild()
                  && activeMdiChild()->textCursor().hasSelection());
      ui->actionCut->setEnabled(hasSelection);
      ui->actionCopy->setEnabled(hasSelection);

      // 有活动窗口且文档有撤销操作时撤销动作可用
      ui->actionUndo->setEnabled(activeMdiChild()
              && activeMdiChild()->document()->isUndoAvailable());
      // 有活动窗口且文档有恢复操作时恢复动作可用
      ui->actionRedo->setEnabled(activeMdiChild()
              && activeMdiChild()->document()->isRedoAvailable());
    }
```

在更新菜单函数中根据是否有活动子窗口，设置了各个菜单动作是否可用。这里剪切复制操作和撤销恢复操作的设置还要进行特殊情况的判断。

```
    MdiChild * MainWindow::activeMdiChild()
    {
```

```
    // 如果有活动窗口,则将其内的中心部件转换为 MdiChild 类型,没有则直接返回 0
    if (QMdiSubWindow * activeSubWindow = ui->mdiArea->activeSubWindow())
      return qobject_cast<MdiChild *>(activeSubWindow->widget());
    return 0;
}
```

这个函数中使用了 QMdiArea 类的 activeSubWindow()函数来获得多文档区域的活动子窗口,然后使用了 T qobject_cast ( QObject * object )函数来进行类型转换。这个函数是 QObject 类中的函数,它将 object 对象指针转换为 T 类型的对象指针,这里将活动窗口的中心部件 QWidget 类型指针转换为 MdiChild 类型指针。这里的 T 类型必须是直接或者间接继承自 QObject 类,而且在其定义中要有 Q_OBJECT 宏变量。现在运行程序,效果如图 1-4 所示。

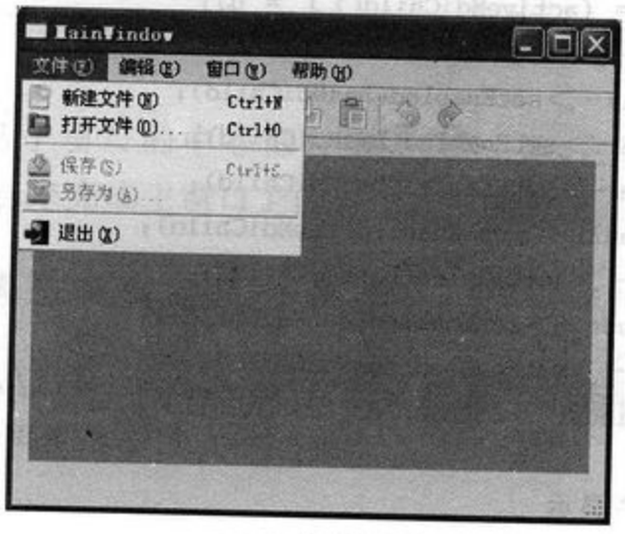



图 1-4　初始化菜单界面

**2. 实现新建文件操作**

首先在 mainwindow.h 文件中添加 private slots:

```
MdiChild *createMdiChild();      //创建子窗口
```

然后在 mainwindow.cpp 文件中添加该槽的定义:

```
MdiChild * MainWindow::createMdiChild()
{
  // 创建 MdiChild 部件
  MdiChild *child = new MdiChild;
  // 向多文档区域添加子窗口,child 为中心部件
  ui->mdiArea->addSubWindow(child);

  // 根据 QTextEdit 类的是否可以复制信号设置剪切复制动作是否可用
  connect(child, SIGNAL(copyAvailable(bool)), ui->actionCut,SLOT(setEnabled(bool)));
  connect(child, SIGNAL(copyAvailable(bool)), ui->actionCopy,SLOT(setEnabled(bool)));
  // 根据 QTextDocument 类的是否可以撤销恢复信号设置撤销恢复动作是否可用
  connect(child->document(), SIGNAL(undoAvailable(bool)),
```

```
        ui->actionUndo, SLOT(setEnabled(bool)));
    connect(child->document(), SIGNAL(redoAvailable(bool)),
        ui->actionRedo, SLOT(setEnabled(bool)));

    return child;
}
```

在这个函数中创建了 MdiChild 部件，并将它作为子窗口的中心部件，然后添加到多文档区域。下面关联了编辑器的信号和我们的菜单动作，让它们可以随着文档的改变而改变状态。最后返回了 MdiChild 对象指针。这里之所以要添加这样一个函数，是因为在下面的打开操作中还要使用到这个函数中的功能，所以将它们从新建文件菜单的触发信号槽中提取出来，另写了这样一个函数。下面更改在上一节中添加的新建文件菜单的触发信号槽：

```
void MainWindow::on_actionNew_triggered()
{
    // 创建 MdiChild
    MdiChild *child = createMdiChild();
    // 新建文件
    child->newFile();
    // 显示子窗口
    child->show();
}
```

因为添加子窗口的操作放到了 createMdiChild()函数中进行，这里只需要调用这个函数就可以了。现在运行程序添加新文件，然后编辑，选中一些字符，可以看到动作状态的变化，如图 1-5 所示。但是，因为现在还没有实现这些动作的功能，所以它们并不可用。

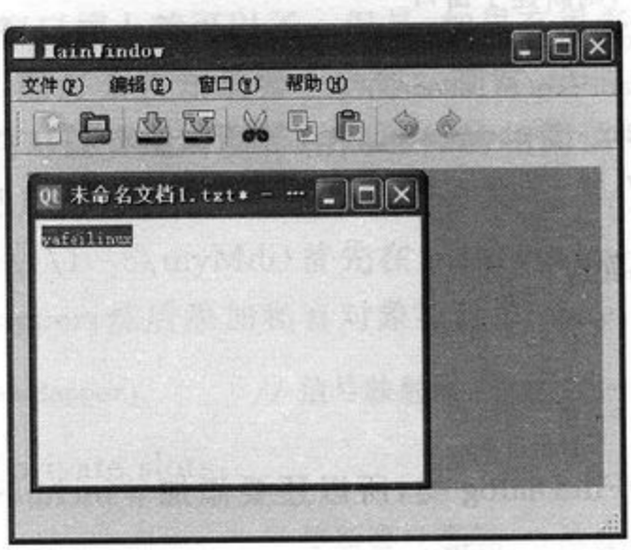



图 1-5　新建文件后菜单状态

### 1.3.2 实现打开文件操作

现在来实现打开文件菜单的功能。当要打开一个文件时，要先判断这个文件是否已经被打开了，这样就需要遍历多文档区域子窗口中的文件，如果发现该文件已经打开，则直接设置该子窗口为活动窗口；否则直接加载要打开的文件，并添加新的子窗口。

（项目源码路径：src\1\1-4\myMdi）首先在 mainwindow.h 文件中先添加类的前置声明 class QMdiSubWindow；然后添加 private 函数声明：

```
QMdiSubWindow * findMdiChild(const QString &fileName);  // 查找子窗口
```

再添加私有槽声明 private slots：

```
void setActiveSubWindow(QWidget * window);              // 设置活动子窗口
```

现在从设计模式进入“打开文件”动作的触发信号 triggered()的槽，更改如下：

```
void MainWindow::on_actionOpen_triggered()
{
  // 获取文件路径
  QString fileName = QFileDialog::getOpenFileName(this);
  // 如果路径不为空，则查看该文件是否已经打开
  if (! fileName.isEmpty()) {
    QMdiSubWindow * existing = findMdiChild(fileName);
    // 如果已经存在，则将对应的子窗口设置为活动窗口
    if (existing) {
      ui->mdiArea->setActiveSubWindow(existing);
      return;
    }
    // 如果没有打开，则新建子窗口
    MdiChild * child = createMdiChild();
    if (child->loadFile(fileName)) {
      ui->statusBar->showMessage(tr("打开文件成功"), 2000);
      child->show();
    } else {
      child->close();
    }
  }
}
```

因为这里使用了 QFileDialog 类，所以还要添加 #include <QFileDialog>头文件。下面是查找子窗口函数的实现：

```
QMdiSubWindow * MainWindow::findMdiChild(const QString &fileName)
{
```

```
    QString canonicalFilePath = QFileInfo(fileName).canonicalFilePath();
    // 利用 foreach 语句遍历子窗口列表,如果其文件路径和要查找的路径相同,则返回该窗口
    foreach (QMdiSubWindow * window, ui->mdiArea->subWindowList()) {
      MdiChild * mdiChild = qobject_cast<MdiChild * >(window->widget());
      if (mdiChild->currentFile() == canonicalFilePath)
        return window;
    }

    return 0;
}
```

这个函数中使用了 foreach 语句来遍历整个多文档区域的所有子窗口,这个函数在《Qt Creator 快速入门》的第 7 章容器类部分讲到。下面是设置活动窗口的实现:

```
void MainWindow::setActiveSubWindow(QWidget * window)
{
    // 如果传递了窗口部件,则将其设置为活动窗口
    if (! window)
        return;
    ui->mdiArea->setActiveSubWindow(qobject_cast<QMdiSubWindow * >(window));
}
```

这个函数的作用就是将传递过来的窗口部件设置为活动窗口。

### 1.3.3　添加子窗口列表

现在为窗口菜单添加显示子窗口列表的功能。我们想每添加一个子窗口就可以在窗口菜单中罗列出它的文件名,而且可以在这个列表中选择一个子窗口,将它设置为活动窗口。这个看似很好实现,只要为窗口菜单添加菜单动作,然后关联这个动作的触发信号到设置活动窗口槽上就可以了。但是,如果有很多个子窗口怎么办,难道要一个一个进行关联吗,那怎么获知是哪个动作?其实,Qt 中提供了一个信号映射器 QSignalMapper 类,它可以实现对多个相同部件的相同信号进行映射,为其添加字符串或者数值参数,然后再发射出去。

(项目源码路径:src\1\1-5\myMdi)首先在 mainwindow.h 文件中添加类的前置声明 class QSignalMapper;然后添加私有对象指针 private:

```
QSignalMapper * windowMapper;          // 信号映射器
```

再添加私有槽声明 private slots:

```
void updateWindowMenu();               // 更新窗口菜单
```

下面到 mainwindow.cpp 文件中添加代码。首先添加 #include <QSignalMapper>头文件,然后在 MainWindow 的构造函数中添加如下代码:

```
// 创建信号映射器
windowMapper = new QSignalMapper(this);
// 映射器重新发射信号,根据信号设置活动窗口
connect(windowMapper, SIGNAL(mapped(QWidget*)),
        this, SLOT(setActiveSubWindow(QWidget*)));

//更新窗口菜单,并且设置当窗口菜单将要显示的时候更新窗口菜单
updateWindowMenu();
connect(ui->menuW, SIGNAL(aboutToShow()), this, SLOT(updateWindowMenu()));
```

这里创建了信号映射器,并且将它的 mapped()信号关联到设置活动窗口槽上,然后更新窗口菜单,并且将窗口菜单的将要显示信号关联到我们的更新菜单槽上,这样每当窗口菜单要显示时都会更新窗口菜单。

```
void MainWindow::updateWindowMenu()
{
  // 先清空菜单,然后再添加各个菜单动作
  ui->menuW->clear();
  ui->menuW->addAction(ui->actionClose);
  ui->menuW->addAction(ui->actionCloseAll);
  ui->menuW->addSeparator();
  ui->menuW->addAction(ui->actionTile);
  ui->menuW->addAction(ui->actionCascade);
  ui->menuW->addSeparator();
  ui->menuW->addAction(ui->actionNext);
  ui->menuW->addAction(ui->actionPrevious);
  ui->menuW->addAction(actionSeparator);

  // 如果有活动窗口,则显示间隔器
  QList<QMdiSubWindow *> windows = ui->mdiArea->subWindowList();
  actionSeparator->setVisible(! windows.isEmpty());

  // 遍历各个子窗口
  for (int i = 0; i < windows.size(); ++i) {
    MdiChild *child = qobject_cast<MdiChild *>(windows.at(i)->widget());
    QString text;
    // 如果窗口数小于9,则设置编号为快捷键
    if (i < 9) {
      text = tr("&%1 %2").arg(i + 1)
                         .arg(child->userFriendlyCurrentFile());
    } else {
    text = tr("%1 %2").arg(i + 1)
                      .arg(child->userFriendlyCurrentFile());
    }
    // 添加动作到菜单,设置动作可以选择
```

```
        QAction * action = ui->menuW->addAction(text);
        action->setCheckable(true);
        // 设置当前活动窗口动作为选中状态
        action->setChecked(child == activeMdiChild());
        // 关联动作的触发信号到信号映射器的 map()槽,这个槽会发射 mapped()信号
        connect(action, SIGNAL(triggered()), windowMapper, SLOT(map()));
        // 将动作与相应的窗口部件进行映射,
        // 在发射 mapped()信号时就会以这个窗口部件为参数
        windowMapper->setMapping(action, windows.at(i));
    }
}
```

更新窗口菜单函数中,先清空了窗口菜单动作,然后再动态添加。这里遍历了多文档区域的各个子窗口,然后以它们中的文件名为文本创建了动作,并将这些动作添加到窗口菜单中。我们将动作的触发信号关联到信号映射器的 map()槽上,然后设置了动作与其对应的子窗口之间的映射,这样触发菜单时就会执行 map()函数,而它又会发射 mapped()信号,这个 mapped()函数会以子窗口部件为参数,因为在构造函数中设置了这个信号与 setActiveSubWindow()函数的关联,所以最终会执行设置活动子窗口函数,并且设置选择的动作指定的子窗口为活动窗口。这时运行程序,效果如图 1-6 所示。


图 1-6　子窗口列表

## 1.3.4　实现其他菜单功能

下面来实现其他一些菜单的功能。因为在前面已经把核心的功能都实现了,而且像剪切、复制等常用功能,QTextEdit 类已经提供了,所以这里只需要调用相应的函数即可。

(项目源码路径:src\1\1-6\myMdi)因为保存、另存为操作在 MdiChild 类中已经实现了,这里只需要调用相应的函数即可。下面是保存菜单动作的触发信号槽的

内容，另存为操作与其相似，这里不再列出。

```
void MainWindow::on_actionSave_triggered()
{
  if(activeMdiChild() && activeMdiChild()->save())
    ui->statusBar->showMessage(tr("文件保存成功"),2000);
}
```

撤销、恢复、剪切、复制和粘贴这些功能函数都由 QTextEdit 类提供，因为 MdiChild 类继承自该类，所以可以直接使用，下面是撤销操作的代码：

```
void MainWindow::on_actionUndo_triggered()
{
  if(activeMdiChild()) activeMdiChild()->undo();
}
```

而窗口菜单中的几个菜单动作，在 QMdiArea 类中都提供了相应的实现函数。下面是关闭菜单的实现：

```
void MainWindow::on_actionClose_triggered()
{
  ui->mdiArea->closeActiveSubWindow();
}
```

最后是帮助菜单中的关于和关于 Qt 两个菜单动作，创建相应的对话框即可。鉴于篇幅限制，这里对这些菜单实现进行了省略，读者只需要在设计模式，进入相应动作的触发信号 triggered()的槽，然后添加代码即可。因为很简单，所以不再讲解，如果出现问题，可以下载我们的源码，该部分源码在 1-6 目录中。最终的运行效果，如图 1-7 所示。

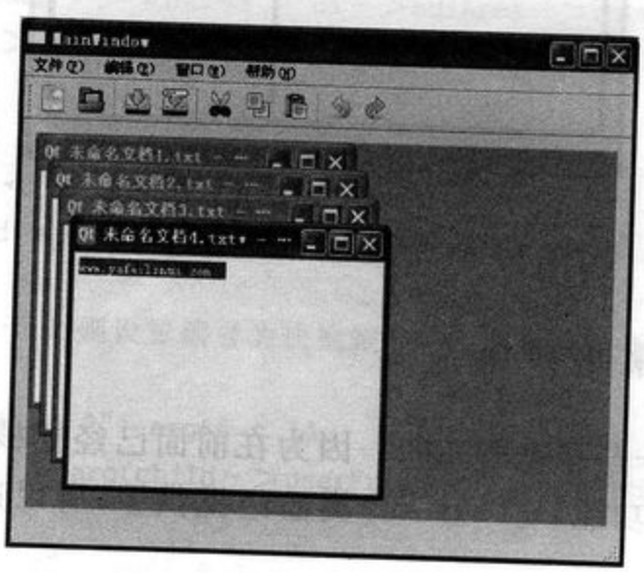



图 1-7　实现其他菜单功能

## 1.4　完善程序功能

应用程序在实现了主要的功能后，还要进行一些必要的设置，使它成为一个完善的应用程序。下面进行几个方面的优化。

### 1.4.1　保存窗口设置

我们都希望自己的应用程序很友好，那么能保存用户对窗口的一些设置（比如大小、位置）等就显得很必要了。Qt 中的 QSettings 类提供了平台无关的永久保存应用程序设置的方法。

（项目源码路径：src\1\1 - 7\myMdi）首先在 mainwindow. h 文件中添加 private 函数声明：

```
void readSettings();                    // 读取窗口设置
void writeSettings();                   // 写入窗口设置
```

然后再添加 protected 函数声明：

```
void closeEvent(QCloseEvent * event);   // 关闭事件
```

到 mainwindow. cpp 添加代码。先添加头文件：

```
#include <QSettings>
#include <QCloseEvent>
```

然后在 MainWindow 类的构造函数中添加代码：

```
readSettings();                         // 初始窗口时读取窗口设置信息
```

下面是关闭事件处理函数的定义：

```
void MainWindow::closeEvent(QCloseEvent * event)
{
  // 先执行多文档区域的关闭操作
  ui->mdiArea->closeAllSubWindows();
  // 如果还有窗口没有关闭，则忽略该事件
  if (ui->mdiArea->currentSubWindow()) {
    event->ignore();
  } else {
    // 在关闭前写入窗口设置
    writeSettings();
    event->accept();
  }
}
```

可以看到我们是在构造窗口时进行窗口设置的读取，在窗口的关闭事件中进行窗口设置的写入的。下面是窗口设置的写入和读取函数：

```
// 写入窗口设置
void MainWindow::writeSettings()
{
  QSettings settings("yafeilinux", "myMdi");
  // 写入位置信息和大小信息
  settings.setValue("pos", pos());
  settings.setValue("size", size());
}

// 读取窗口设置
void MainWindow::readSettings()
{
  QSettings settings("yafeilinux", "myMdi");
  QPoint pos = settings.value("pos", QPoint(200, 200)).toPoint();
  QSize size = settings.value("size", QSize(400, 400)).toSize();
  move(pos);
  resize(size);
}
```

这两个函数的使用这里不再详细介绍。下面从设计模式进入“退出”菜单动作的触发信号槽，更改如下：

```
void MainWindow::on_actionExit_triggered()
{
  // 等价于 QApplication::closeAllWindows();
  qApp->closeAllWindows();
}
```

这里使用了 qApp 指针，它是 QApplication 对象的全局指针，因为在一个应用程序中只能定义一个 QApplication 对象。现在运行程序，改变窗口位置和大小，然后关闭程序，重新运行，测试一下运行效果。

### 1.4.2 自定义右键菜单

现在运行程序时会发现子窗口中的右键菜单是英文的，需要将它改为中文。下面来自定义右键菜单，这个需要在 MdiChild 类中进行，就是重新实现 QTextEdit 类的上下文菜单事件。首先在 mdichild.h 文件中添加 protected 函数声明：

```
void contextMenuEvent(QContextMenuEvent * e);     // 右键菜单事件
```

然后到 mdichild.cpp 文件中添加头文件 #include <QMenu>，再进行函数的定义：

```
void MdiChild::contextMenuEvent(QContextMenuEvent * e)
{
  // 创建菜单,并向其中添加动作
  QMenu * menu = new QMenu;
  QAction * undo = menu->addAction(tr("撤销(&U)"),this,
                                   SLOT(undo()),QKeySequence::Undo);
  undo->setEnabled(document()->isUndoAvailable());
  QAction * redo = menu->addAction(tr("恢复(&R)"),this,
                                   SLOT(redo()),QKeySequence::Redo);
  redo->setEnabled(document()->isRedoAvailable());
  menu->addSeparator();
  QAction * cut = menu->addAction(tr("剪切(&T)"),this,
                                  SLOT(cut()),QKeySequence::Cut);
  cut->setEnabled(textCursor().hasSelection());
  QAction * copy = menu->addAction(tr("复制(&C)"),this,
                                   SLOT(copy()),QKeySequence::Copy);
  copy->setEnabled(textCursor().hasSelection());
  menu->addAction(tr("粘贴(&P)"),this,SLOT(paste()),QKeySequence::Paste);
  QAction * clear = menu->addAction(tr("清空"),this,SLOT(clear()));
  clear->setEnabled(! document()->isEmpty());
  menu->addSeparator();
  QAction * select = menu->addAction(tr("全选"),this,
                     SLOT(selectAll()),QKeySequence::SelectAll);
  select->setEnabled(! document()->isEmpty());

  // 获取鼠标的位置,然后在这个位置显示菜单
  menu->exec(e->globalPos());

  // 最后销毁这个菜单
  delete menu;
}
```

在这个函数中创建了一个菜单,然后给它添加了一些动作;在添加动作时,直接为这些动作指定了触发信号 triggered()的槽。在为这些动作添加快捷键时,我们使用了 QKeySequence 类提供的默认操作快捷键。这里还为一些动作设置了是否可用的条件。最后在鼠标指针位置弹出菜单,并在执行结束时销毁这个菜单。运行程序,测试一下效果,如图 1-8 所示。

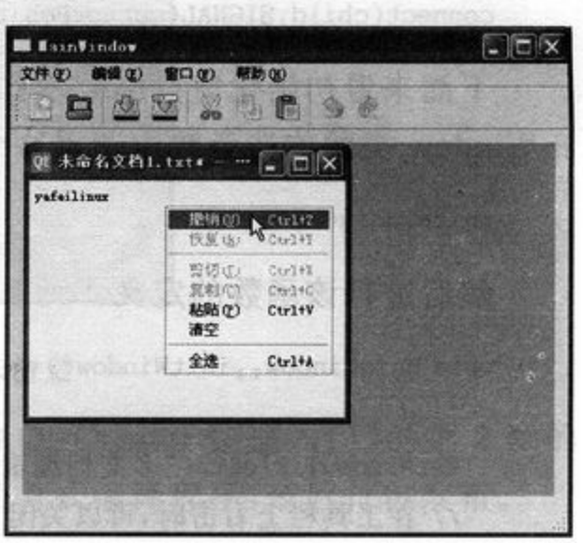



图 1-8　自定义右键菜单

### 1.4.3 其他功能

最后还想要在状态栏中可以显示编辑器中光标所在的行号和列号，然后设置窗口的标题和状态栏的一些显示。首先在 mainwindow.h 文件中添加私有槽的声明 private slots：

```
void showTextRowAndCol();          // 显示文本的行号和列号
```

然后添加一个私有函数的声明 private：

```
void initWindow();                 // 初始化窗口
```

先来看显示光标位置函数的定义：

```
void MainWindow::showTextRowAndCol()
{
  // 如果有活动窗口，则显示其中光标所在的位置
  if(activeMdiChild()){
    // 因为获取的行号和列号都是从 0 开始的，所以我们这里进行了加 1
    int rowNum = activeMdiChild()->textCursor().blockNumber() + 1;
    int colNum = activeMdiChild()->textCursor().columnNumber() + 1;

    ui->statusBar->showMessage(tr("%1 行 %2 列")
                               .arg(rowNum).arg(colNum),2000);
  }
}
```

在这个函数中获取了活动窗口中光标的位置，并在状态栏中显示，为了每次编辑器中的光标位置变化时都可以调用这个函数，需要在 createMdiChild()函数中的“return child;”一行代码前添加一行代码：

```
// 每当编辑器中的光标位置改变，就重新显示行号和列号
connect(child,SIGNAL(cursorPositionChanged()),this,SLOT(showTextRowAndCol()));
```

下面来看初始化窗口函数。首先添加头文件#include <QLabel>，然后在 MainWindow 类的构造函数中调用这个函数，即在最后添加代码：

```
initWindow();
```

然后进行该函数的定义：

```
void MainWindow::initWindow()
{
  setWindowTitle(tr("多文档编辑器"));
  // 在工具栏上右击时，可以关闭工具栏
  ui->mainToolBar->setWindowTitle(tr("工具栏"));
```

```
    // 当多文档区域的内容超出可视区域后,出现滚动条
    ui->mdiArea->setHorizontalScrollBarPolicy(Qt::ScrollBarAsNeeded);
    ui->mdiArea->setVerticalScrollBarPolicy(Qt::ScrollBarAsNeeded);

    ui->statusBar->showMessage(tr("欢迎使用多文档编辑器"));
    QLabel *label = new QLabel(this);
    label->setFrameStyle(QFrame::Box | QFrame::Sunken);
    label->setText(
      tr("<a href=\"http://www.yafeilinux.com\">yafeilinux.com</a>"));
    // 标签文本为富文本
    label->setTextFormat(Qt::RichText);
    // 可以打开外部链接
    label->setOpenExternalLinks(true);
    ui->statusBar->addPermanentWidget(label);
    ui->actionNew->setStatusTip(tr("创建一个文件"));
    // 这里省略了其他动作的状态提示
    ......
}
```

这里设置了窗口的标题和工具栏的标题,然后为多文档区域设置了滚动条,添加了网站的链接。最后设置了各个动作的状态提示信息,将鼠标移动到这些动作上时,在状态栏会显示这些提示信息。这里只是列举了新建文件菜单动作的状态提示,关于其他动作的提示,可以自己编写,或者查看我们的源码。最终运行效果如图 1-9 所示。

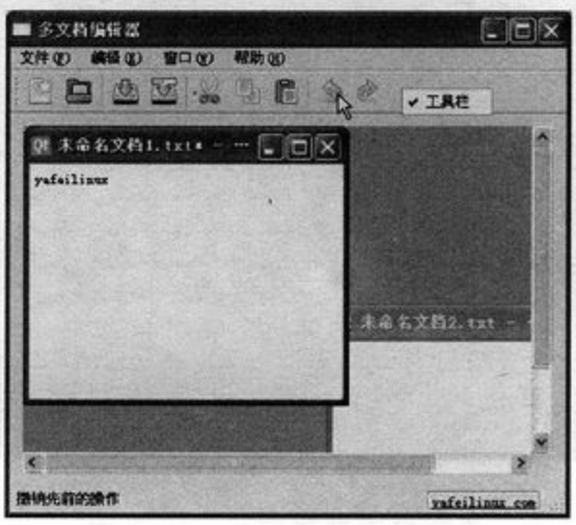



图 1-9　状态栏提示信息

到这里,整个程序就设计完成了。可以按照《Qt Creator 快速入门》的第 2 章的内容给这个程序添加程序图标,然后以 Release 的方式编译程序,最后打包发布。

## 1.5 小 结

这个多文档编辑器程序只是实现了编辑器的一些最基本功能，还有很多功能没能实现，比如富文本的处理、打印和查找等。不过作为初学者的第一个综合实例，它已经包含了太多的知识点，如果要做一个功能强大的编辑器，那么这一章也许要写几百页。如果还有兴趣扩展这个程序，可以看一下 Qt 中提供的 Text Edit 示例，或者去我们网站上查看多文档编辑器开源软件，那个程序更加综合。

读者应该认真学习这一章，不仅是学习其中的知识点，更多的是学习一种方法，编写综合程序的方法。可以看到，我们程序的功能是一点一点加上去的，再庞大的程序也是将功能模块一个个加上去的，不要设想一下子就写出一个功能强大的应用程序。而且在程序编写过程中，一定会出现各种问题，不要气馁，不要烦躁，因为这是正常现象，学会多使用 qDebug()函数。

# 第 2 章　方块游戏

这一章将讲述一个俄罗斯方块游戏的实例程序，其综合应用了《Qt Creator 快速入门》中图形动画篇和影音应用篇的多个知识点。因为该方块游戏主要基于《Qt Creator 快速入门》中第 11 章讲述的图形视图框架，所以建议学完该章再来学习本章。Qt 中提供了 Tetrix 和 QSTetrix 两个版本的俄罗斯方块游戏，前者是基于基本窗口部件和 2D 绘图的，而后者是使用 Qt Script 来实现的，它们分别位于 Widgets 和 QtScript 分类中。如果先参考过了 Tetrix 示例程序，再来学习本章的方块游戏，那么一定会感叹图形视图框架的强大，使用它无需再考虑那些复杂的算法，而只需要考虑游戏的逻辑和实现效果即可。方块游戏的最终运行效果如图 2-1 所示。

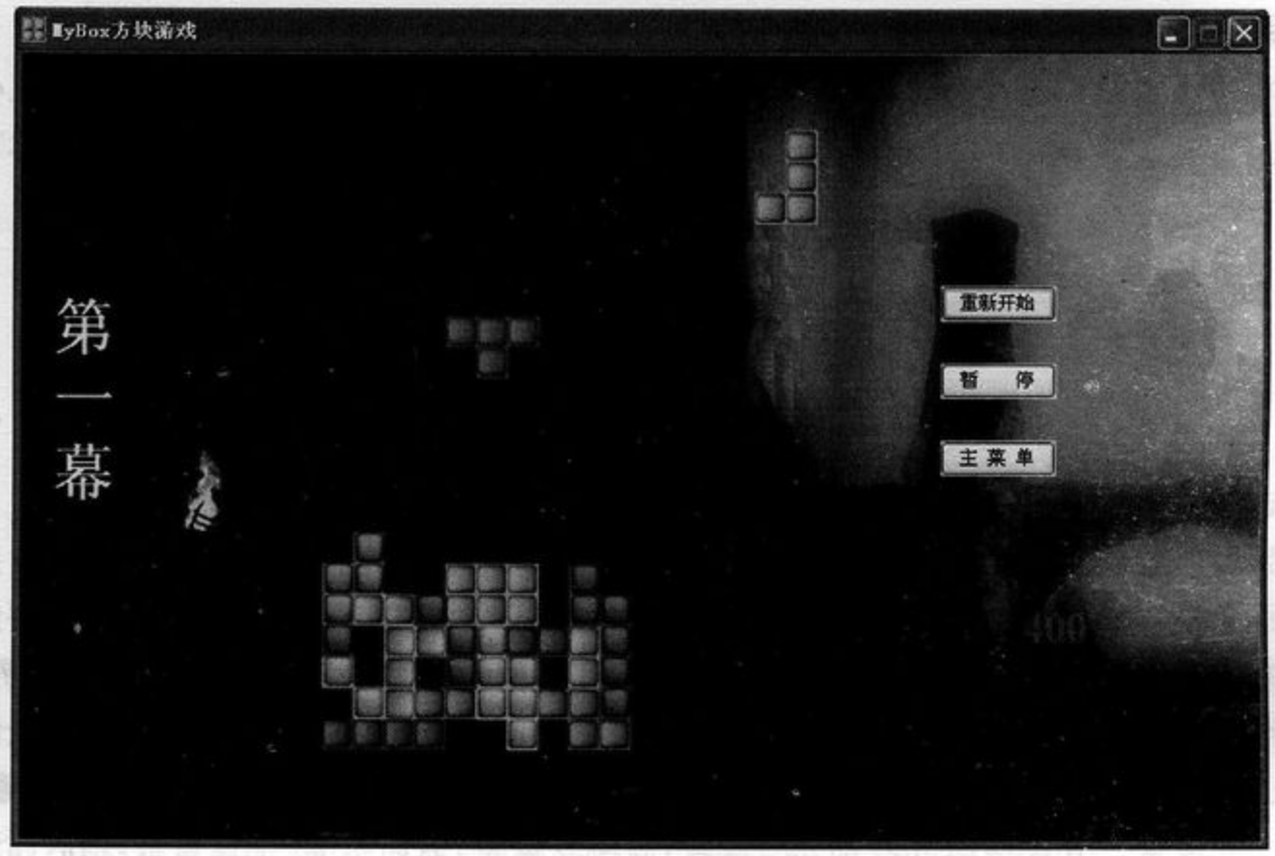



图 2-1　方块游戏运行效果

## 2.1　方块游戏架构

本实例设计的是一个标准的俄罗斯方块游戏，我们有必要先来了解一下俄罗斯方块游戏本身的一些内容。俄罗斯方块(Tetris)是一款游戏机上的经典游戏，它由

俄罗斯人发明，所以称作俄罗斯方块。游戏的规则就是对自动输出并向下移动的各种形状的方块进行移动、旋转，然后排列成完整的一行并被消除，根据消除的行数来获取分数。在游戏中，有一个区域用来摆放方块，该区域宽为 10，高为 20，以小正方形为单位，它可以看作是拥有 20 行 10 列的一个网格。标准的游戏中一共有 7 种方块，它们都是由 4 个小正方形组成的规则图形，依据形状分别用字母 I、J、L、O、S、T 和 Z 来命名。

这里使用了图形视图框架来实现整个游戏的设计。小正方形由 OneBox 类来表示，它继承自 QGraphicsObject 类，之所以继承自这个类，是因为这样就可以使用信号和槽机制，还可以使用属性动画。小正方形就是一个宽和高都为 20 像素的正方形图形项。游戏中的方块图形由方块组 BoxGroup 类来实现，多继承自 QObject 和 QGraphicsItemGroup 类，这样该类也可以使用信号和槽机制。方块组是一个宽和高都是 80 像素的图形项组，其中包含了 4 个小方块，通过设置小方块的位置来实现 7 种标准的方块图形，它们的形状和位置如图 2-2 所示。在 BoxGroup 类中实现了方块图形的创建、移动和碰撞检测。

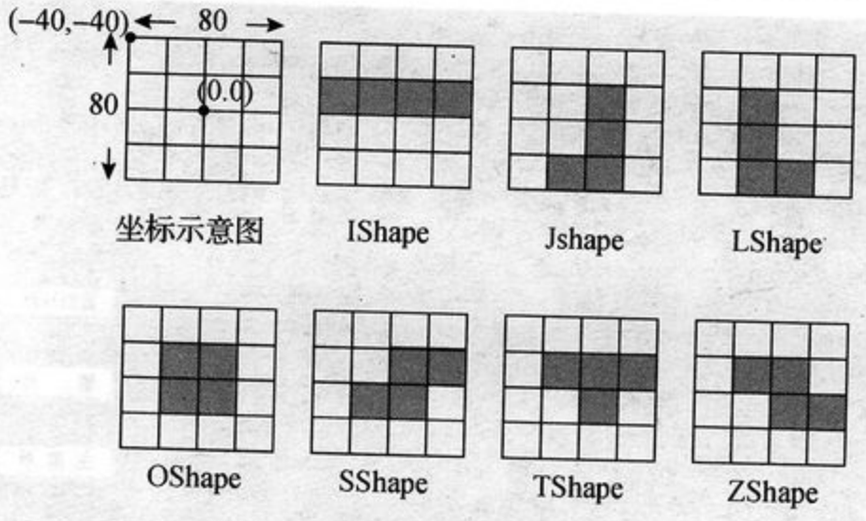



图 2-2　7 种方块图形

方块移动区域在游戏场景中使用 4 条直线图形项所围区域来表示，之所以要这样实现，是因为这样可以通过方块组是否与直线图形项碰撞来检测是否移动出界。整个游戏界面由 MyView 类来实现，该类继承自 QGraphicsView 类，实现了场景设置、游戏逻辑设置和游戏声音设置及其他所有控制功能。整个游戏场景宽 800 像素，高 500 像素。方块移动区域宽 200 像素，高 400 像素，纵向每 20 个像素被视作一行，所以共有 20 行；横行也是每 20 个像素视作一列，所以共有 10 列，该区域可以看作一个由 20 行 10 列 20×20 像素的方格组成的网格。方块组在方块移动区域的初始位置为上方正中间，但方块组的最上方一行小正方形在方块移动区域以外，这样可以保证方块组完全出现在移动区域的最上方，方块组每移动一次，就是移动一个方格的位置。在场景中还设置了下一个要出现方块的提示方块、游戏暂停等控制按钮和游戏分数级别的显示文本，整个场景的示意图如图 2-3 所示。

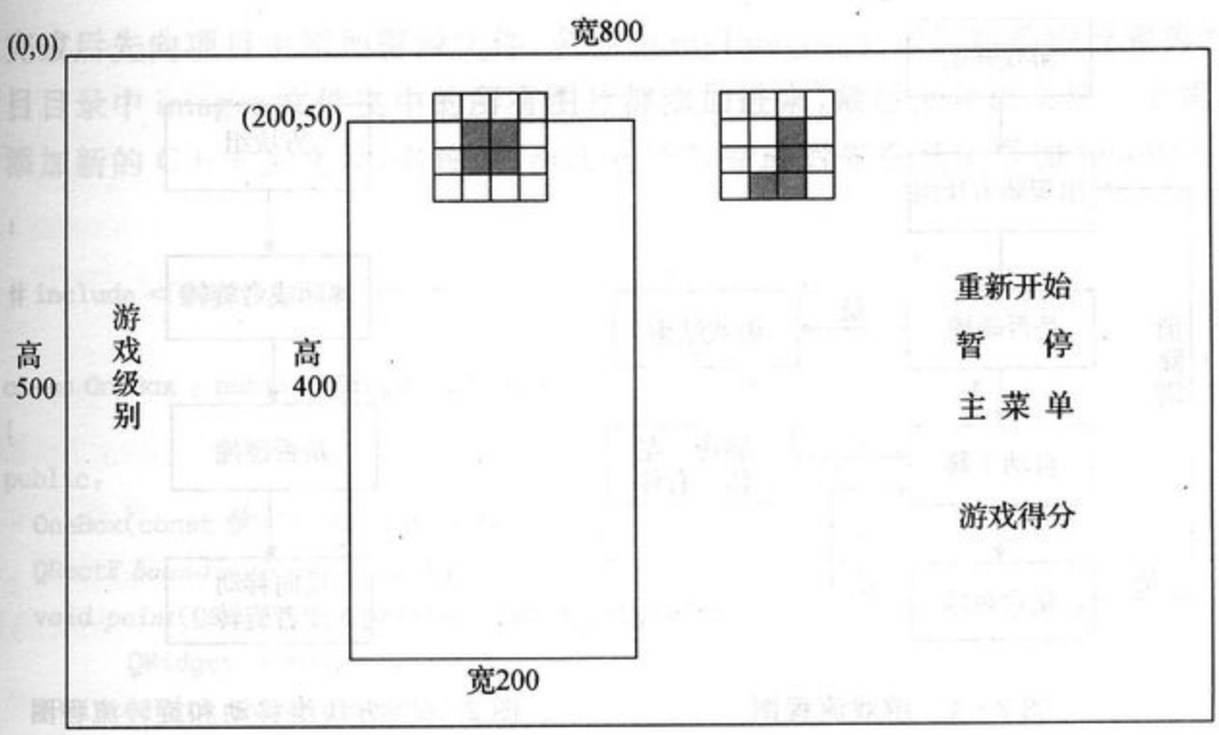



图 2-3　游戏场景示意图

总的来说，本实例由 OneBox、BoxGroup 和 MyView 这 3 个类构成，分别实现了小正方形、方块图形和游戏场景，在后面的内容中我们所说的小方块是指 OneBox，方块组是指 BoxGroup，游戏场景是指 MyView。

## 2.2　实现游戏逻辑

### 2.2.1　游戏流程

方块游戏的简单流程如图 2-4 所示。当开始游戏后，首先创建一个新的方块组，并将其添加到场景中的方块移动区域上方。然后进行碰撞检测，如果这时已经发生了碰撞，那么游戏结束；如果没有发生碰撞，就可以使用键盘的方向键对其进行旋转变形或者左右移动。当到达指定时间时方块组会自动下移一个方格，这时再次判断是否发生碰撞，如果发生了碰撞，先消除满行的方格，然后出现新的方块组，并继续进行整个流程。其中方块组的移动、旋转、碰撞检测等都在 BoxGroup 类中进行；游戏的开始、结束、出现新的方块组、消除满行等都在 MyView 类中进行。

**1. 方块组的移动和旋转**

方块组的左移、右移、下移和旋转都是先进行该操作，然后判断是否发生碰撞，如果发生了碰撞就再进行反向操作。比如，使用方向键左移方块组，那么就先将方块组左移一格，然后进行碰撞检测，看是否与边界线或者其他方块碰撞了，如果发生了碰撞，那么就再移回来，即右移一格。方块组的移动和旋转流程如图 2-5 所示。

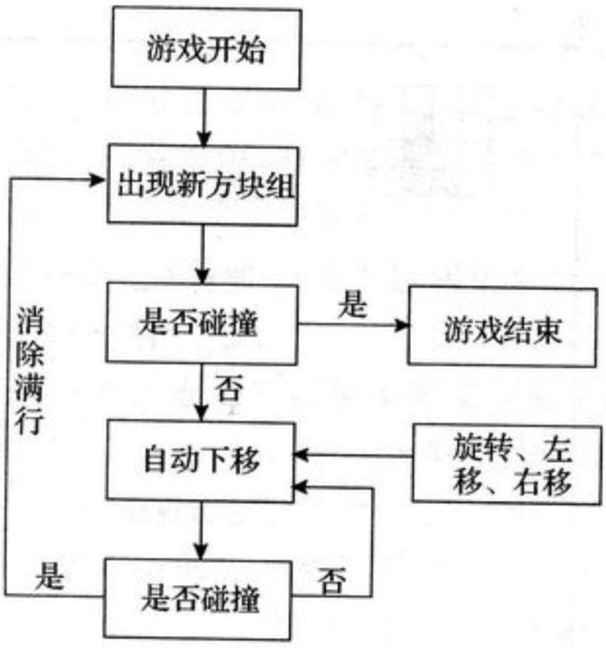



图 2-4　游戏流程图


图 2-5　方块组移动和旋转流程图

**2. 碰撞检测**

对于方块组的碰撞检测，其实是使用方块组中的 4 个小方块来进行的，这样就不用再为每个方块图形都设置一个碰撞检测时使用的形状。要进行碰撞检测时，对每一个小方块都使用函数来获取与它们碰撞的图形项的数目，因为现在小方块在方块组中，所以应该只有方块组与它们碰撞了（由于我们对小方块的形状进行了设置，所以挨着的 4 个小方块相互间不会被检测出发生了碰撞，这个在后面的程序中会看到），也就是说与它们碰撞的图形项数目应该不会大于 1，如果有哪个小方块发现与它碰撞的图形项的数目大于 1，那么说明已经发生了碰撞。

**3. 游戏结束**

当一个新的方块组出现时，就立即对其进行碰撞检测，如果它一出现就与其他方块发生了碰撞，说明游戏已经结束，这时由方块组发射游戏结束信号。

**4. 消除满行**

游戏开始后，每当出现一个新的方块以前，都判断游戏移动区域的每一行是否已经拥有 10 个小方块。如果有一行已经拥有了 10 个小方块，说明该行已满，那么就销毁该行的所有小方块，然后让该行上面的所有小方块都下移一格。

## 2.2.2　实现基本游戏功能

下面先创建项目，然后往项目中分别添加 OneBox、BoxGroup 和 MyView 等类，从而实现最基本的游戏功能。

**1. 设计小方块**

（项目源码路径：src\2\2-1\myGame）新建空的 Qt 项目，项目名称为 myGame。

建立完成后先向项目中添加资源文件，名称为 myImages，完成后将前缀设置为空，并将项目目录中 images 文件夹中的所有图片都添加进来，最后保存该文件。下面向项目中添加新的 C++头文件，名称为“mybox.h”，完成后先在其中添加 OneBox 类的定义：

```
#include <QGraphicsObject>

class OneBox : public QGraphicsObject
{
public:
  OneBox(const QColor &color = Qt::red);
  QRectF boundingRect() const;
  void paint(QPainter * painter, const QStyleOptionGraphicsItem * option,
        QWidget * widget);
  QPainterPath shape() const;
private:
  QColor brushColor;
};
```

然后向项目中添加新的 C++源文件，名称为“mybox.cpp”，完成后，向其中添加 OneBox 类的实现代码：

```
#include "mybox.h"
#include <QPainter>

OneBox::OneBox(const QColor &color) : brushColor(color)
{
}
QRectF OneBox::boundingRect() const
{
  qreal penWidth = 1;
  return QRectF(-10 - penWidth / 2, -10 - penWidth / 2,
              20 + penWidth, 20 + penWidth);
}
void OneBox::paint(QPainter * painter, const QStyleOptionGraphicsItem * option, QWid-
get * widget)
{
  // 为小方块使用贴图
  painter->drawPixmap(-10, -10, 20, 20, QPixmap(":/images/box.gif"));
  painter->setBrush(brushColor);
  QColor penColor = brushColor;
  // 将颜色的透明度减小
  penColor.setAlpha(20);
  painter->setPen(penColor);
  painter->drawRect(-10, -10, 20, 20);
```

```
}
// 形状比边框矩形小0.5像素,这样方块组中的小方块才不会发生碰撞
QPainterPath OneBox::shape() const
{
  QPainterPath path;
  path.addRect(-9.5, -9.5, 19, 19);
  return path;
}
```

paint()函数先绘制了宽和高均为20的正方形图片,然后又在它的上面使用指定的画刷颜色绘制了一个小正方形,当指定的画刷颜色为透明颜色时,就会在底层绘制的贴图上绘制一层颜色,这样来实现不同颜色的小方块。之所以要使用贴图,是为了使小方块看起来很有质感。而将画笔颜色的透明度减小,是为了使小方块的边界线不会和填充的画刷颜色一样。

为了在使用碰撞检测函数时不会将方块组中两个相邻的小方块检测为发生了碰撞,这里将小方块的形状设置为比实际形状小0.5像素。这样即便两个小方块看起来是挨着的,因为检测碰撞时是使用shape()返回的形状,所以它们不会被检测为发生了碰撞。

**2. 设计方块组**

下面再到mybox.h文件中添加头文件#include <QGraphicsItemGroup>,然后添加BoxGroup类的定义:

```
class BoxGroup : public QObject, public QGraphicsItemGroup
{
  Q_OBJECT
public:
  enum BoxShape {IShape, JShape, LShape, OShape, SShape,
                 TShape, ZShape, RandomShape};
  BoxGroup();
  QRectF boundingRect() const;
  bool isColliding();
  void createBox(const QPointF &point = QPointF(0, 0),
                 BoxShape shape = RandomShape);
  void clearBoxGroup(bool destroyBox = false);
  BoxShape getCurrentShape() {return currentShape;}
protected:
  void keyPressEvent(QKeyEvent * event);
signals:
  void needNewBox();
  void gameFinished();
public slots:
  void moveOneStep();
```

```
    void startTimer(int interval);
    void stopTimer();
private:
    BoxShape currentShape;
    QTransform oldTransform;
    QTimer * timer;
};
```

这里使用枚举变量 BoxShape 定义了 8 个方块形状，最后一个 RandomShape 是随机形状，如果指定了该形状，那么就在前 7 个形状中随机选取一个。

下面到 mybox.cpp 文件中添加头文件：

```
#include <QKeyEvent>
#include <QTimer>
```

然后添加 BoxGroup 类的实现代码：

```
BoxGroup::BoxGroup()
{
    setFlags(QGraphicsItem::ItemIsFocusable);
    // 保存变换矩阵，当 BoxGroup 进行旋转后，可以使用它来进行恢复
    oldTransform = transform();
    timer = new QTimer(this);
    connect(timer, SIGNAL(timeout()), this, SLOT(moveOneStep()));
    currentShape = RandomShape;
}
QRectF BoxGroup::boundingRect() const
{
    qreal penWidth = 1;
    return QRectF(-40 - penWidth / 2, -40 - penWidth / 2,
                  80 + penWidth, 80 + penWidth);
}
```

这里的构造函数中先保存了方块组的变换矩阵，这是因为如果在后面的操作中对方块组进行了旋转，方块组的变换矩阵就会改变，当需要设置新的方块形状时就会出现位置错误。这里还设置了定时器，是用来进行方块组自动下移的。下面添加键盘事件处理函数的定义：

```
void BoxGroup::keyPressEvent(QKeyEvent * event)
{
    switch (event->key())
    {
    case Qt::Key_Down :
        moveBy(0, 20);
        if (isColliding()) {
            moveBy(0, -20);
```

```
            // 将小方块从方块组中移除到场景中
            clearBoxGroup();
            // 需要显示新的方块
            emit needNewBox();
        }
        break;
    case Qt::Key_Left :
        moveBy( - 20, 0);
        if (isColliding())
            moveBy(20, 0);
        break;
    case Qt::Key_Right :
        moveBy(20, 0);
        if (isColliding())
            moveBy( - 20, 0);
        break;
    case Qt::Key_Up :
        rotate(90);
        if(isColliding())
            rotate( - 90);
        break;
    // 空格键实现坠落
    case Qt::Key_Space :
        moveBy(0, 20);
        while (! isColliding()) {
            moveBy(0, 20);
        }
        moveBy(0, - 20);
        clearBoxGroup();
        emit needNewBox();
        break;
    }
}
```

这里分别使用键盘的上下左右4个方向键实现了旋转、下移、左移和右移操作，使用空格键实现了坠落操作，就是将方块一次移动到最下方。进行这些操作时都是先进行操作，然后进行碰撞检测，如果发生碰撞，则进行反向操作来恢复到以前的状态。在下移和坠落操作时，如果发生碰撞，说明已经移动到底，就要将方块组中的小方块移动到场景中，然后发射needNewBox()信号，告知需要一个新的方块组。下面是碰撞检测函数的定义：

```
bool BoxGroup::isColliding()
{
    QList<QGraphicsItem * > itemList = childItems();
```

```
    QGraphicsItem * item;
    foreach (item, itemList)
    {
      if(item->collidingItems().count() > 1)
        return true;
    }
    return false;
}
```

这里就是使用方块组中的 4 个小方块来进行碰撞检测的，判断与它们碰撞的图形项的数目，如果超过 1，表明已经发生了碰撞。下面是用于删除方块组中的小方块的函数的定义：

```
void BoxGroup::clearBoxGroup(bool destroyBox)
{
  QList<QGraphicsItem * > itemList = childItems();
  QGraphicsItem * item;
  foreach (item, itemList) {
    removeFromGroup(item);
    if (destroyBox) {
      OneBox * box = (OneBox * ) item;
      box->deleteLater();
    }
  }
}
```

这里就是使用 QGraphicsItemGroup 的 removeFromGroup()函数将小方块从方块组中移动到场景中。这个函数还有一个参数，如果设置该参数为真，那么不但要从方块组中移除小方块，还要将这些小方块销毁掉。下面添加用于创建方块的函数：

```
void BoxGroup::createBox(const QPointF &point, BoxShape shape)
{
  static const QColor colorTable[7] = {
          QColor(200, 0, 0, 100), QColor(255, 200, 0, 100),
          QColor(0, 0, 200, 100), QColor(0, 200, 0, 100),
          QColor(0, 200, 255, 100), QColor(200, 0, 255, 100),
          QColor(150, 100, 100, 100)
  };
  int shapeID = shape;
  if (shape == RandomShape) {
    // 产生 0-6 之间的随机数
    shapeID = qrand() % 7;
  }
  QColor color = colorTable[shapeID];
  QList<OneBox * > list;
```

```
    //恢复方块组的变换矩阵
    setTransform(oldTransform);
    for (int i = 0; i < 4; ++i) {
        OneBox *temp = new OneBox(color);
        list << temp;
        addToGroup(temp);
    }
    switch (shapeID) {
    case IShape:
        currentShape = IShape;
        list.at(0)->setPos(-30, -10);
        list.at(1)->setPos(-10, -10);
        list.at(2)->setPos(10, -10);
        list.at(3)->setPos(30, -10);
        break;
    case JShape:
        currentShape = JShape;
        list.at(0)->setPos(10, -10);
        list.at(1)->setPos(10, 10);
        list.at(2)->setPos(-10, 30);
        list.at(3)->setPos(10, 30);
        break;
    case LShape:
        currentShape = LShape;
        list.at(0)->setPos(-10, -10);
        list.at(1)->setPos(-10, 10);
        list.at(2)->setPos(-10, 30);
        list.at(3)->setPos(10, 30);
        break;
    case OShape:
        currentShape = OShape;
        list.at(0)->setPos(-10, -10);
        list.at(1)->setPos(10, -10);
        list.at(2)->setPos(-10, 10);
        list.at(3)->setPos(10, 10);
        break;
    case SShape:
        currentShape = SShape;
        list.at(0)->setPos(10, -10);
        list.at(1)->setPos(30, -10);
        list.at(2)->setPos(-10, 10);
        list.at(3)->setPos(10, 10);
        break;
    case TShape:
        currentShape = TShape;
```

```
    list.at(0)->setPos(-10, -10);
    list.at(1)->setPos(10, -10);
    list.at(2)->setPos(30, -10);
    list.at(3)->setPos(10, 10);
    break;
  case ZShape :
    currentShape = ZShape;
    list.at(0)->setPos(-10, -10);
    list.at(1)->setPos(10, -10);
    list.at(2)->setPos(10, 10);
    list.at(3)->setPos(30, 10);
    break;
  default : break;
  }
  // 设置位置
  setPos(point);
  // 如果开始就发生碰撞,说明已经结束游戏
  if (isColliding()) {
    stopTimer();
    emit gameFinished();
  }
}
```

在这个函数中有一个颜色表,先按照给定的方块形状来获取一个颜色,然后使用这个颜色来创建 4 个小方块,再按指定的形状来设置 4 个小方块在方块组中的位置。注意:小方块的位置是指小方块中心(0,0)点的位置。当设置好方块形状后,将方块组放到场景中指定的位置处,然后进行碰撞检测,如果发生了碰撞,说明游戏已经结束,那么就发射游戏结束的 gameFinished()信号。其中的 qrand()函数是用来产生随机数的,关于随机数的知识,可以参考《Qt Creator 快速入门》中 6.4 节。下面添加与自动下移有关的 3 个函数:

```
// 开启定时器
void BoxGroup::startTimer(int interval)
{
  timer->start(interval);
}
// 向下移动一步
void BoxGroup::moveOneStep()
{
  moveBy(0, 20);
  if (isColliding()) {
    moveBy(0, -20);
    clearBoxGroup();
    emit needNewBox();
```

```
    }
  }
  // 停止定时器
  void BoxGroup::stopTimer()
  {
    timer->stop();
  }
```

它们就是用来开启定时器和关闭定时器，然后等定时器溢出时下移一格，moveOneStep()函数与向下方向键进行的操作是一样的。关于定时器的知识，也可以参考《Qt Creator 快速入门》中 6.4 节。

**3. 添加游戏场景**

下面添加 MyView 类。首先向项目中添加新的 C++类，类名为 MyView，基类为 QGraphicsView，类型信息选择“继承自 QWidget”。完成后到 myview.h 文件中更改 MyView 类的定义如下：

```
class BoxGroup;

class MyView : public QGraphicsView
{
  Q_OBJECT
public:
  explicit MyView(QWidget *parent = 0);
public slots:
  void startGame();
  void clearFullRows();
  void moveBox();
  void gameOver();
private:
  BoxGroup *boxGroup;
  BoxGroup *nextBoxGroup;
  QGraphicsLineItem *topLine;
  QGraphicsLineItem *bottomLine;
  QGraphicsLineItem *leftLine;
  QGraphicsLineItem *rightLine;
  qreal gameSpeed;
  QList<int> rows;
  void initView();
  void initGame();
  void updateScore(const int fullRowNum = 0);
};
```

然后到 myview.cpp 文件中，添加 MyView 类的实现：

```
#include "myview.h"
```

```
#include "mybox.h"
#include <QIcon>
// 游戏的初始速度
static const qreal INITSPEED = 500;

MyView::MyView(QWidget * parent) :
  QGraphicsView(parent)
{
  initView();
}
// 初始化游戏界面
void MyView::initView()
{
  // 使用抗锯齿渲染
  setRenderHint(QPainter::Antialiasing);
  // 设置缓存背景,这样可以加快渲染速度
  setCacheMode(CacheBackground);
  setWindowTitle(tr("MyBox 方块游戏"));
  setWindowIcon(QIcon(":/images/icon.png"));
  setMinimumSize(810, 510);
  setMaximumSize(810, 510);
  // 设置场景
  QGraphicsScene * scene = new QGraphicsScene;
  scene->setSceneRect(5, 5, 800, 500);
  scene->setBackgroundBrush(QPixmap(":/images/background.png"));
  setScene(scene);
  // 方块可移动区域的 4 条边界线
  topLine = scene->addLine(197, 47, 403, 47);
  bottomLine = scene->addLine(197,453,403,453);
  leftLine = scene->addLine(197,47,197,453);
  rightLine = scene->addLine(403,47,403,453);
  // 当前方块组和提示方块组
  boxGroup = new BoxGroup;
  connect(boxGroup, SIGNAL(needNewBox()), this, SLOT(clearFullRows()));
  connect(boxGroup, SIGNAL(gameFinished()), this, SLOT(gameOver()));
  scene->addItem(boxGroup);
  nextBoxGroup = new BoxGroup;
  scene->addItem(nextBoxGroup);

  startGame();
}
```

这里将方块可移动区域的 4 条边界线都向外面扩展了 3 个像素,这样可以保证小方块在边界处再向外移动一格就会与边界线相交,从而检测出它们发生了碰撞。然后分别创建了两个方块组,一个是在方块区域中的当前方块组,一个是用来提示下

一个要出现的方块组。最后调用了startGame()槽来开始游戏，下面添加该槽的定义：

```
// 开始游戏
void MyView::startGame()
{
  initGame();
}
```

这个槽现在只是调用了initGame()函数来初始化游戏，下面添加initGame()函数的定义：

```
// 初始化游戏
void MyView::initGame()
{
  boxGroup->createBox(QPointF(300, 70));
  boxGroup->setFocus();
  boxGroup->startTimer(INITSPEED);
  gameSpeed = INITSPEED;
  nextBoxGroup->createBox(QPointF(500, 70));
}
```

初始化游戏时，分别创建了当前方块组和提示方块组，然后设置当前方块组获得焦点，这样它就可以被键盘控制了。因为在多处要用到初始速度，我们使用了全局变量INITSPEED来表示初始的游戏速度，它是以毫秒为单位的，而gameSpeed变量是用来保存当前游戏速度的，它在暂停游戏返回时会被用到。下面添加清空满行槽的定义：

```
// 清空满行
void MyView::clearFullRows()
{
  // 获取比一行方格较大的矩形中包含的所有小方块
  for (int y = 429; y > 50; y -= 20) {
    QList<QGraphicsItem *> list = scene()->items(199, y, 202, 2, Qt::Contain-
sItemShape);
    // 如果该行已满
    if (list.count() == 10) {
      foreach (QGraphicsItem *item, list) {
        OneBox *box = (OneBox *) item;
        box->deleteLater();
      }
      // 保存满行的位置
      rows << y;
    }
  }
```

```
    // 如果有满行,下移满行上面的各行再出现新的方块组
    // 如果没有满行,则直接出现新的方块组
    if(rows.count() > 0) {
      moveBox();
    } else {
      boxGroup->createBox(QPointF(300, 70), nextBoxGroup->getCurrentShape());
      // 清空并销毁提示方块组中的所有小方块
      nextBoxGroup->clearBoxGroup(true);
      nextBoxGroup->createBox(QPointF(500, 70));
    }
  }
```

这里使用了比方块移动区域的一行方块所形成的矩形大一个像素的矩形来检测该行中包含的所有小方块的数量,如果数量为 10,说明该行已满,那么就使用 QObject 类的 deleteLater()来销毁该行的所有小方块。然后使用 QList 对象 rows 保存所有满行的位置,如果满行数量大于 0,那么就使用 moveBox()下移满行上面的所有小方块;如果没有出现满行,则使用提示方块组中的方块形状来为当前方块组生成新形状的方块图形,然后再为提示方块组也生成一个新形状的方块图形。下面添加 moveBox()槽的定义:

```
  // 下移满行上面的所有小方块
  void MyView::moveBox()
  {
    // 从位置最靠上的满行开始
    for (int i = rows.count(); i > 0; --i) {
      int row = rows.at(i - 1);
      foreach (QGraphicsItem *item, scene()->items(199, 49, 202, row - 47, Qt::Contains
ItemShape)) {
        item->moveBy(0, 20);
      }
    }
    // 更新分数
    updateScore(rows.count());
    // 将满行列表清空为 0
    rows.clear();
    // 等所有行下移以后再出现新的方块组
    boxGroup->createBox(QPointF(300, 70), nextBoxGroup->getCurrentShape());
    nextBoxGroup->clearBoxGroup(true);
    nextBoxGroup->createBox(QPointF(500, 70));
  }
```

这里需要分别将每一个满行上面的所有小方块都下移一格,这需要从最上面的满行开始。然后获取从方块移动区域上边界到当前行的位置所形成的矩形,使用比这个矩形大一个像素的矩形来获取当前行以上的所有小方块,让它们分别下移一个

方格。当完成所有移动后，通过使用 updateScore()函数来更新分数显示。最后将 rows 的内容清空，并出现新的方块组。

下面添加更新分数显示函数和游戏结束槽的定义，它们的具体实现到后面再进行添加。

```
void MyView::updateScore(const int fullRowNum)
{
  // 更新分数
}
void MyView::gameOver()
{
  // 游戏结束
}
```

**4. 添加主函数**

最后再向项目中添加“main.cpp”文件，更改内容如下：

```
#include <QApplication>
#include "myview.h"
#include <QTextCodec>
#include <QTime>
int main(int argc, char * argv[])
{
  QApplication app(argc, argv);
  QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
  // 设置随机数的初始值
  qsrand(QTime(0, 0, 0).secsTo(QTime::currentTime()));
  MyView view;
  view.show();
  return app.exec();
}
```

此时运行程序，已经可以实现基本的游戏功能了。可以看到，程序中并没有使用复杂的算法，而只是使用了图形项组和碰撞检测等内容，这就是得益于图形视图框架的支持。

## 2.3 游戏优化

### 2.3.1 添加满行销毁动画

为了实现更加绚丽的游戏效果，我们为销毁满行的小方块添加动画效果。通过使用动画框架，为图形项添加动画是非常容易的。

(项目源码路径:src\2\2 - 2\myGame)首先在 myview.cpp 文件中添加头文件:

```
#include <QPropertyAnimation>
#include <QGraphicsBlurEffect>
#include <QTimer>
```

然后在 clearFullRows()槽中,将调用 deleteLater()的一行代码更改为:

```
QGraphicsBlurEffect *blurEffect = new QGraphicsBlurEffect;
box->setGraphicsEffect(blurEffect);
QPropertyAnimation *animation = new QPropertyAnimation(box, "scale");
animation->setEasingCurve(QEasingCurve::OutBounce);
animation->setDuration(250);
animation->setStartValue(4);
animation->setEndValue(0.25);
animation->start(QAbstractAnimation::DeleteWhenStopped);
connect(animation, SIGNAL(finished()), box, SLOT(deleteLater()));
```

这里先为小方块使用了模糊图形效果,然后为其添加了放大再缩小的属性动画,等动画执行结束后才调用 deleteLater()槽。图形效果在《Qt Creator 快速入门》中 11.3.1 小节讲到,而属性动画在 11.4.1 小节讲到。然后再将后面代码中调用 moveBox()函数的一行代码更改为:

```
QTimer::singleShot(400, this, SLOT(moveBox()));
```

这里使用了只执行一次的定时器,其目的是等待所有小方块都销毁后再移动满行上面的小方块。现在运行程序,当销毁满行时已经可以出现动画效果了,如图 2 - 6 所示。也可以更改这里的代码来实现自己的动画特效。

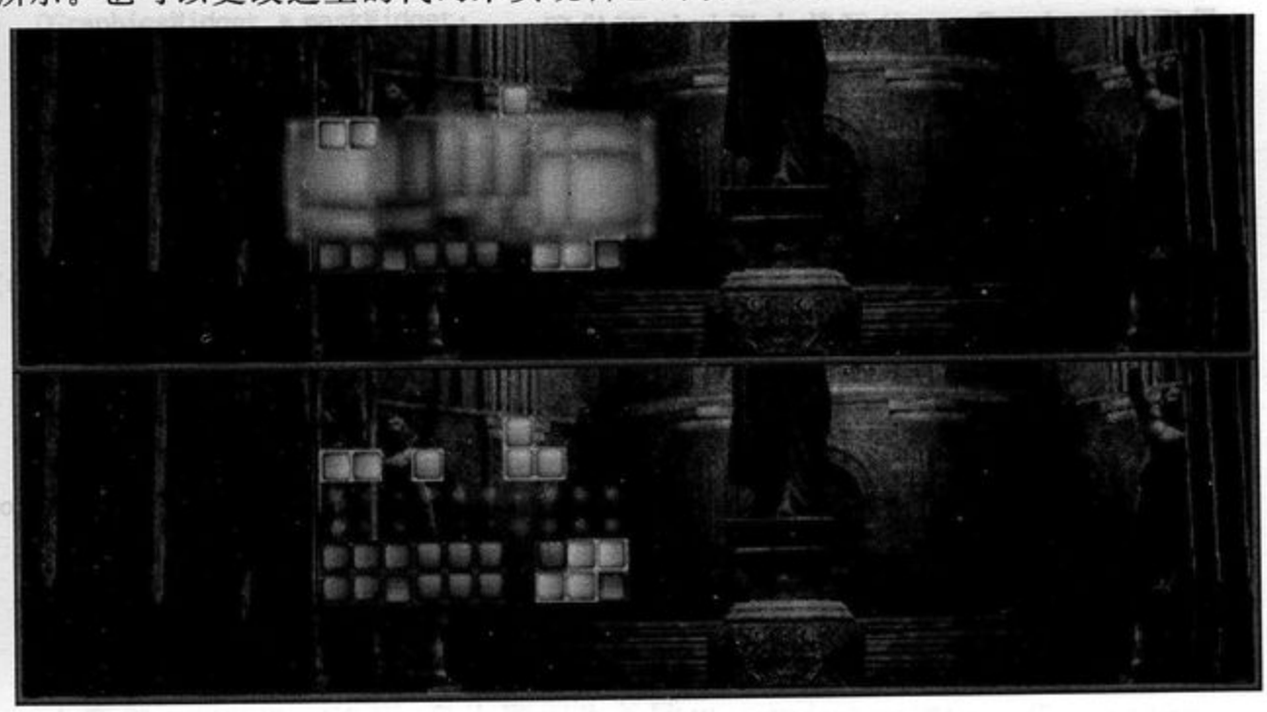


图 2 - 6　销毁满行动画效果

### 2.3.2 添加游戏级别设置

游戏级别是根据所获得的分数而改变的，所以可以在updateScore()函数中进行游戏级别的设置。

(项目源码路径：src\2\2-3\myGame)先在myview.h文件中添加两个文本图形项对象的定义，它们用来显示分数和级别信息。在private中继续添加如下代码：

```
QGraphicsTextItem * gameScoreText;
QGraphicsTextItem * gameLevelText;
```

然后在myview.cpp文件中的initView()函数中调用startGame()槽的前面添加如下代码：

```
// 得分文本
gameScoreText = new QGraphicsTextItem(0, scene);
gameScoreText->setFont(QFont("Times", 20, QFont::Bold));
gameScoreText->setPos(650, 350);
// 级别文本
gameLevelText = new QGraphicsTextItem(0, scene);
gameLevelText->setFont(QFont("Times", 30, QFont::Bold));
gameLevelText->setPos(20, 150);
```

再到initGame()中添加如下代码：

```
scene()->setBackgroundBrush(QPixmap(":/images/background01.png"));
gameScoreText->setHtml(tr("<font color=red>0</font>"));
gameLevelText->setHtml(tr("<font color=white>第<br>一<br>幕</font>"));
```

最后到updateScore()函数中添加如下代码：

```
int score = fullRowNum * 100;
int currentScore = gameScoreText->toPlainText().toInt();
currentScore += score;
// 显示当前分数
gameScoreText->setHtml(tr("<font color=red>%1</font>").arg(currentScore));
// 判断级别
if (currentScore < 500) {
  // 第一级，什么都不用做
} else if (currentScore < 1000) { // 第二级
  gameLevelText->setHtml(tr("<font color=white>第<br>二<br>幕</font
>"));
  scene()->setBackgroundBrush(QPixmap(":/images/background02.png"));
  gameSpeed = 300;
  boxGroup->stopTimer();
  boxGroup->startTimer(gameSpeed);
} else {
```

```
    // 添加下一个级别的设置
}
```

这里使用销毁的满行数目来设置分数，再使用分数来设置级别，在每一个级别中使用不同的背景图片和游戏速度。还可以继续添加更多的级别设置。现在运行程序就可以显示分数和级别了，当分数超过 500 后就会更改级别和背景图片。

### 2.3.3　添加游戏控制按钮和面板

一个完善的游戏应该有一个清晰的游戏控制逻辑，比如当打开程序后应该出现一个主菜单；可以通过单击开始按钮来开始游戏；可以使用暂停按钮暂停游戏；使用重新开始游戏按钮来开始新的游戏；当游戏结束时应该可以提示要重新开始游戏或者结束游戏返回主菜单等。另外还设计了一个黑色透明的遮罩部件，当出现暂停等特殊情况时来遮罩整个场景；还有一个选项面板，用来进行一些选项设置；以及一个帮助面板，用来显示帮助信息。

(项目源码路径：src\2\2 - 4\myGame)首先在 myview. h 文件中添加几个槽的声明：

```
void restartGame();
void finishGame();
void pauseGame();
void returnGame();
```

然后添加一些私有对象的定义：

```
// 遮罩面板
QGraphicsWidget * maskWidget;
// 各种按钮
QGraphicsWidget * startButton;
QGraphicsWidget * finishButton;
QGraphicsWidget * restartButton;
QGraphicsWidget * pauseButton;
QGraphicsWidget * optionButton;
QGraphicsWidget * returnButton;
QGraphicsWidget * helpButton;
QGraphicsWidget * exitButton;
QGraphicsWidget * showMenuButton;
// 各种文本
QGraphicsTextItem * gameWelcomeText;
QGraphicsTextItem * gamePausedText;
QGraphicsTextItem * gameOverText;
```

下面到 myview. cpp 中，首先添加头文件：

```
# include <QPushButton>
```

```
#include <QGraphicsProxyWidget>
#include <QApplication>
#include <QLabel>
#include <QFileInfo>
```

然后在initView()函数中,先删除startGame()的调用代码,然后添加如下代码:

```
// 设置初始为隐藏状态
topLine->hide();
bottomLine->hide();
leftLine->hide();
rightLine->hide();
gameScoreText->hide();
gameLevelText->hide();
// 黑色遮罩
QWidget *mask = new QWidget;

……省略了一些代码
```

因为要使用开始按钮来控制游戏的开始,所以这里先删除了startGame()的调用。然后设置了一些图形项在开始时处于隐藏状态。后面就是一些部件和图形项的创建与设置代码,这里为了节省篇幅省略了这些代码。注意:读者需要从网站上下载本书的源码,将这里省略的代码补充完整后再进行后面的内容。

下面在startGame()中调用initGame()函数的代码前面添加如下代码:

```
gameWelcomeText->hide();
startButton->hide();
optionButton->hide();
helpButton->hide();
exitButton->hide();
maskWidget->hide();
```

然后在initGame()中添加如下代码:

```
restartButton->show();
pauseButton->show();
showMenuButton->show();
gameScoreText->show();
gameLevelText->show();
topLine->show();
bottomLine->show();
leftLine->show();
rightLine->show();
// 可能以前返回主菜单时隐藏了 boxGroup
boxGroup->show();
```

下面在游戏结束gameOver()槽中添加如下代码:

```
pauseButton->hide();
showMenuButton->hide();
maskWidget->show();
gameOverText->show();
restartButton->setPos(370, 200);
finishButton->show();
```

下面再添加其他几个槽的定义：

```
// 重新开始游戏
void MyView::restartGame()
{
  ……省略了该部分代码
}
// 结束当前游戏
void MyView::finishGame()
{
  ……省略了该部分代码
}
// 暂停游戏
void MyView::pauseGame()
{
  ……省略了该部分代码
}
// 返回游戏,处于暂停状态时
void MyView::returnGame()
{
  ……省略了该部分代码
}
```

在这些槽中就是进行一些按钮的显隐和一些函数的调用，这里就不再对这些槽的实现进行讲解，可以参考源码。现在运行程序，主菜单界面如图 2-7 所示，游戏暂停界面如图 2-8 所示。

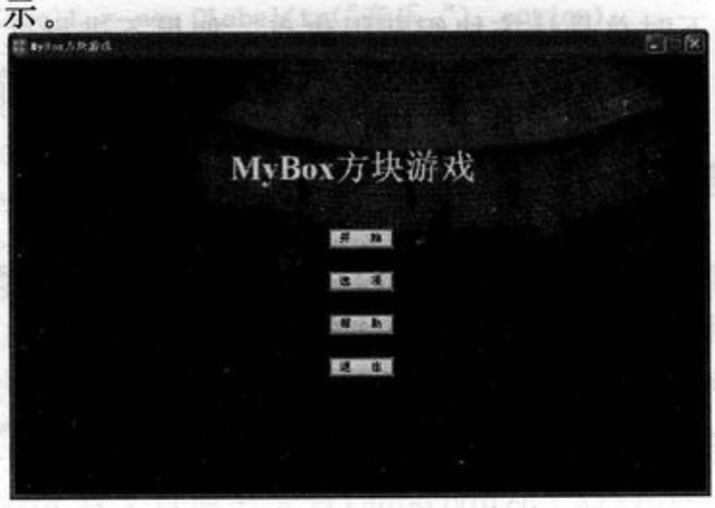



图 2-7　主菜单界面


图2-8 游戏暂停界面

为了进行游戏时总是当前方块组获得焦点，也就是说，当在进行游戏状态，键盘按键总能控制方块组进行移动，我们需要重写一下视图的键盘按下事件处理函数。在myview.h文件中添加如下代码：

```
protected:
  void keyPressEvent(QKeyEvent * event);
```

然后到myview.cpp文件添加该函数的定义如下：

```
void MyView::keyPressEvent(QKeyEvent * event)
{
  // 如果正在进行游戏，当键盘按下时总是方块组获得焦点
  if (pauseButton->isVisible())
    boxGroup->setFocus();
  else
    boxGroup->clearFocus();
  QGraphicsView::keyPressEvent(event);
}
```

这里使用了pauseButton是否可见来判断是否处于游戏状态，如果处于游戏状态，那么当有键盘按下时总是让方块组获得焦点。如果不进行这样的设置，当单击了场景的其他地方时，焦点就会改变，就无法使用键盘控制方块组了。

### 2.3.4 添加背景音乐和音效

为了使游戏更加完善，我们为其添加了背景音乐和音效，音效主要是销毁满行动画的声音。这里使用了《Qt Creator快速入门》中第14章讲述的Phonon多媒体框架来实现声音的播放和控制，对于背景音乐和音效分别使用了一个媒体对象来控制，还创建了两个音量滑块部件来控制它们的音量大小，这两个部件生成在了选项面板上。

（项目源码路径：src\2\2-5\myGame）首先在项目文件myGame.pro中添加如下一行代码：

```
QT += phonon
```

完成后保存项目文件。然后到 myview.h 文件中添加头文件：

```
#include <phonon>
```

再添加一个槽声明：

```
void aboutToFinish();
```

最后添加两个私有对象定义：

```
Phonon::MediaObject *backgroundMusic;
Phonon::MediaObject *clearRowSound;
```

这里的 backgroundMusic 用来控制背景音乐，clearRowSound 用来控制满行销毁动画的声音。下面到 myview.cpp 文件中，先添加一个指定声音文件路径的全局变量，这样可以方便更改声音文件的路径：

```
static const QString SOUNDPATH = "../myGame/sounds/";
```

然后到 initView()函数的最后添加如下代码：

```
// 设置声音
backgroundMusic = new Phonon::MediaObject(this);
clearRowSound = new Phonon::MediaObject(this);
Phonon::AudioOutput *audio1 = new Phonon::AudioOutput(Phonon::MusicCategory,
this);
Phonon::AudioOutput *audio2 = new Phonon::AudioOutput(Phonon::MusicCategory,
this);
Phonon::createPath(backgroundMusic, audio1);
Phonon::createPath(clearRowSound, audio2);
// 设置音量控制部件，它们显示在选项面板上
Phonon::VolumeSlider *volume1 = new Phonon::VolumeSlider(audio1, option);
Phonon::VolumeSlider *volume2 = new Phonon::VolumeSlider(audio2, option);
QLabel *volumeLabel1 = new QLabel(tr("音乐:"), option);
QLabel *volumeLabel2 = new QLabel(tr("音效:"), option);
volume1->move(100, 100);
volume2->move(100, 200);
volumeLabel1->move(60, 105);
volumeLabel2->move(60, 205);
connect(backgroundMusic, SIGNAL(aboutToFinish()), this, SLOT(aboutToFinish()));
// 因为播放完毕后会进入暂停状态，再调用 play()将无法进行播放
// 需要在播放完毕后使其进入停止状态
connect(clearRowSound, SIGNAL(finished()), clearRowSound, SLOT(stop()));
backgroundMusic->setCurrentSource(Phonon::MediaSource(SOUNDPATH
                                  + "background.mp3"));
clearRowSound->setCurrentSource(Phonon::MediaSource(SOUNDPATH
```

```
                                                    + "clearRow.mp3"));
    backgroundMusic->play();
```

这里就是创建了媒体对象，然后生成了媒体图，并且创建了音量控制部件，最后为媒体对象设置了媒体源。下面再到initGame()函数中添加如下代码：

```
    backgroundMusic->setCurrentSource(Phonon::MediaSource(SOUNDPATH
                                                    + "background01.mp3"));
    backgroundMusic->play();
```

这样在开始游戏时就会更改播放的背景音乐了。下面设置在执行销毁满行动画时播放指定的声音，在clearFullRows()函数中singleShot()函数调用前添加如下一行代码：

```
    clearRowSound->play();
```

当结束游戏返回主菜单时，要更改背景音乐，在finishGame()函数中添加如下代码：

```
    backgroundMusic->setCurrentSource(Phonon::MediaSource(SOUNDPATH
                                                    + "background.mp3"));
    backgroundMusic->play();
```

当改变了游戏级别时也要更改背景音乐，在updateScore()函数中调用startTimer()的代码后面添加如下代码：

```
    if (QFileInfo(backgroundMusic->currentSource().fileName()).baseName()!=
"background02") {
        backgroundMusic->setCurrentSource(Phonon::MediaSource(SOUNDPATH + "back-
ground02.mp3"));
        backgroundMusic->play();
    }
```

最后添加aboutToFinish()槽的定义：

```
// 背景音乐将要播放完毕时继续重新播放
void MyView::aboutToFinish()
{
  backgroundMusic->enqueue(backgroundMusic->currentSource());
}
```

因为媒体对象当前的媒体源播放结束后，如果媒体对象的队列中没有后续的媒体源，那么就会处于暂停状态。我们希望一首背景音乐播放结束后可以重复进行播放，所以要在播放将要结束时将当前媒体源加入到播放队列中。现在运行程序，就可以实现音乐的播放了，在选项面板中还可以设置音量的大小。

### 2.3.5　添加程序启动画面

当一个应用程序需要很长的启动时间时，往往会先显示一个启动画面。在 Qt 中提供了 QSplashScreen 类来实现启动画面，它使用起来很简单，只需要在主函数中添加几行代码即可。

继续在前面的程序中添加代码。在 main.cpp 文件中添加头文件：

```
#include <QSplashScreen>
```

然后在主函数中创建 view 对象前添加如下代码：

```
QPixmap pix(":/images/logo.png");
QSplashScreen splash(pix);
splash.resize(pix.size());
splash.show();
app.processEvents();
```

这里将启动画面的大小设置为图片的大小。调用 processEvents()函数是为了使程序在显示启动画面的同时仍然能够响应鼠标等事件。下面在调用 show()的代码后面添加如下代码：

```
splash.finish(&view);
```

finish()函数用来在指定窗口初始化完成后，结束启动画面。此时运行程序，会发现主界面出现以前会在屏幕中心出现一幅启动画面。

到这里，整个方块游戏就设计完成了。这个实例中实现了基本的方块游戏功能，设计了绚丽的背景和动画效果，而且还实现了背景音乐和音效。不过，对于保存游戏进度、显示分数排行等功能，这里并没有实现，这些功能读者可以自己尝试着实现。这个方块游戏是基于图形视图框架的，所以游戏中的很多设计难点都被简化了，如果想使用一般的方法来实现方块游戏，可以参考一下我们网站上的方块游戏教程。

## 2.4　小　结

学习完本章，读者应该能进一步掌握 Qt 中图形视图框架的应用，尤其是碰撞检测、图形项组的应用。也应该学会在图形视图框架中使用动画框架来实现动画效果，还要学会将 Phonon 多媒体框架应用到其他程序中以实现音乐的播放和控制。

# 第 3 章　音乐播放器

这一章将讲述一个音乐播放器的实例程序，它是对《Qt Creator 快速入门》的基础应用篇、图形动画篇、影音应用篇和数据应用篇中多个章节知识点的综合应用。其中，音乐播放使用了《Qt Creator 快速入门》中第 14 章讲述的 Phonon 多媒体框架；播放列表使用了第 16 章讲述的 QTableWidget 部件；LRC 歌词解析使用了第 7 章讲述的 QMap 容器类以及正则表达式；桌面歌词使用了第 10 章讲述的 2D 绘图的部分内容；系统托盘图标使用了 QSystemTrayIcon 类。

因为音乐播放器的核心功能使用了《Qt Creator 快速入门》中第 14 章讲述的 Phonon 多媒体框架，所以建议学完该章再来学习本章。该实例是基于 Qt 中的 Music Player 示例程序的，在 Phonon 分类下可以找到。音乐播放器的最终运行效果如图 3-1 所示。

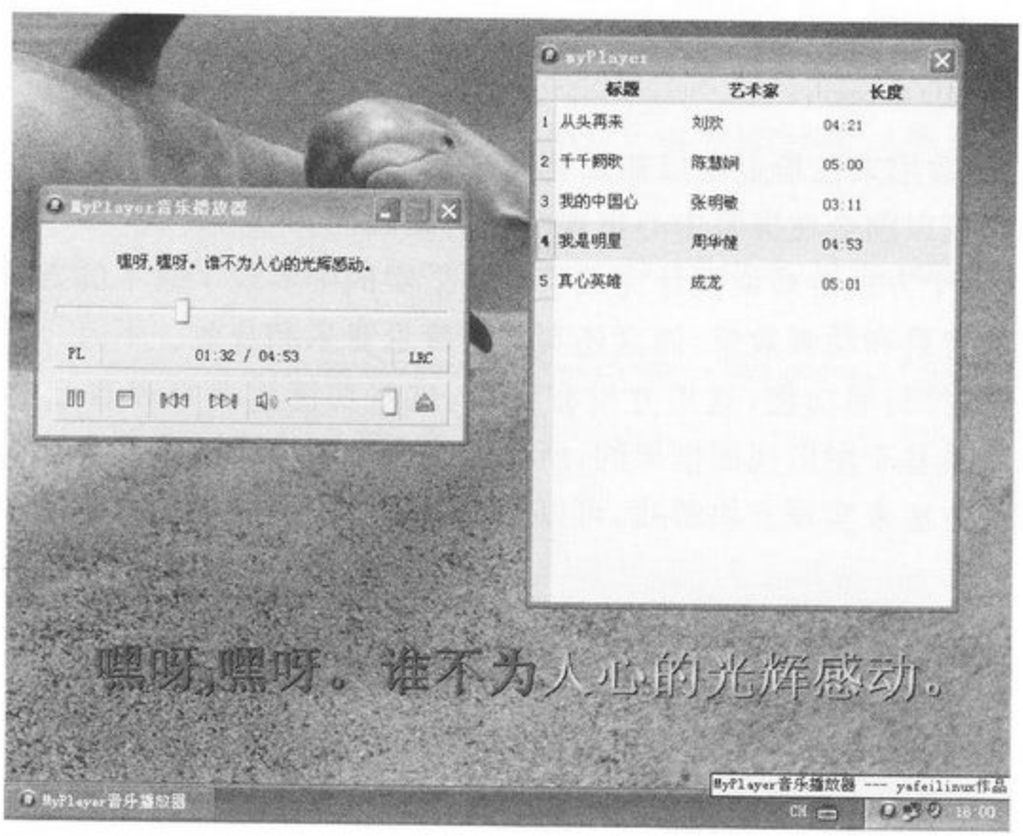



图 3-1　音乐播放器运行效果

## 3.1　播放器整体架构

将设计的音乐播放器取名为 MyPlayer，主要由 3 部分组成：播放器主界面、播放

列表窗口和桌面歌词部件。其中，播放器主界面由 MyWidget 类来实现，继承自 QWidget 类，主要用来实现音乐的播放控制、获取歌曲文件信息并构建播放列表窗口、解析歌词文件并构建桌面歌词部件；播放列表窗口由 MyPlaylist 类来实现，继承自 QTableWidget 类，主要用来显示播放列表；桌面歌词部件由 MyLrc 类来实现，继承自 QLabel 类，主要用来将 MyWidget 传递过来的歌词文本在桌面上使用漂亮的文字显示出来。

播放器设计的流程如图 3-2 所示。简单过程描述如下：在播放器中打开 MP3 等音乐文件后，会对这些文件进行解析，获得歌曲的标题、艺术家和长度等信息，然后在播放列表中显示出来。可以单击播放列表中的一个歌曲来进行播放，并使用 Phonon::MediaObject 对象来进行音乐播放的控制。当要播放一个歌曲时，就解析该歌曲对应的 LRC 歌词文件，如果该歌词文件存在，就在播放时根据播放时间把相应的歌词文本传递给桌面歌词部件，从而实现了歌词的动态显示。

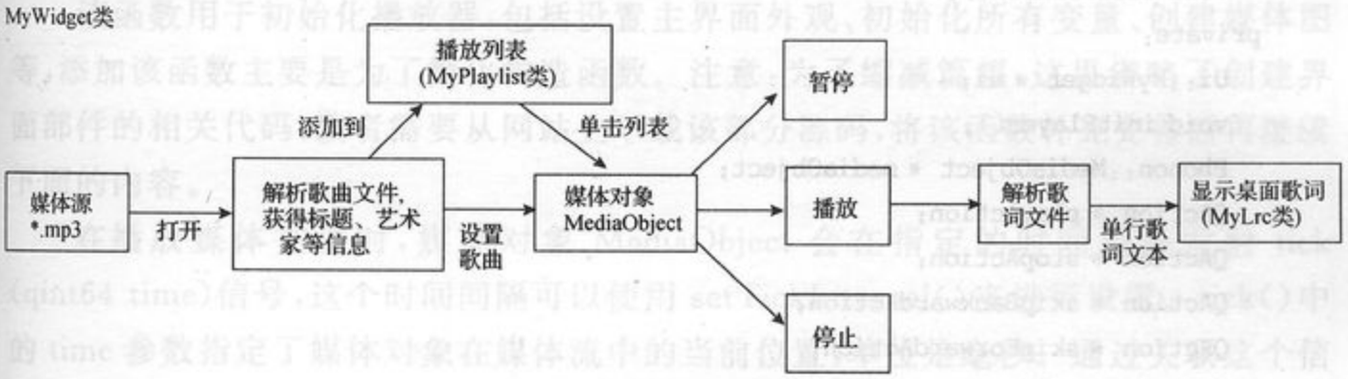



图 3-2　播放器设计流程图

## 3.2　实现音乐播放

这一节实现播放器最基本的音乐播放功能，主要是设计 MyWidget 类。因为这里是使用 Phonon 多媒体框架来实现媒体播放的，所以需要先学习《Qt Creator 快速入门》中第 14 章的相关内容。这里还会使用到第 14 章没有涉及的媒体对象状态的知识。

### 3.2.1　创建播放器主界面

(项目源码路径：src\3\3-1\myPlayer)新建 Qt Gui 应用，项目名称为 myPlayer，基类选择 QWidget，类名改为 MyWidget。完成后先往项目中添加一个资源文件，文件名称为 myImages，前缀设置为空，将项目目录中 images 文件夹里的所有图标添加进来，然后保存该文件即可。完成后在项目文件 myPlayer.pro 中添加代码：

```
QT      += phonon
```

添加完成后保存该文件。下面进入mywidget.h文件，先添加头文件和类的前置声明：

```
#include <Phonon>
class QLabel;
```

然后添加私有槽和私有变量，完成后代码如下：

```
private slots:
    void updateTime(qint64 time);
    void setPaused();
    void skipBackward();
    void skipForward();
    void openFile();
    void setPlaylistShown();
    void setLrcShown();
private:
    Ui::MyWidget *ui;
    void initPlayer();
    Phonon::MediaObject *mediaObject;
    QAction *playAction;
    QAction *stopAction;
    QAction *skipBackwardAction;
    QAction *skipForwardAction;
    QLabel *topLabel;
    QLabel *timeLabel;
```

下面进入mywidget.cpp文件，添加头文件包含：

```
#include <QLabel>
#include <QToolBar>
#include <QVBoxLayout>
#include <QTime>
```

再到构造函数中调用initPlayer()函数来初始化播放器，即添加如下一行代码：

```
initPlayer();
```

下面添加initPlayer()函数的定义：

```
// 初始化播放器
void MyWidget::initPlayer()
{
    // 设置主界面标题、图标和大小
    setWindowTitle(tr("MyPlayer音乐播放器"));
    setWindowIcon(QIcon(":/images/icon.png"));
    setMinimumSize(320, 160);
    setMaximumSize(320, 160);
```

```
    // 创建媒体图
    mediaObject = new Phonon::MediaObject(this);
    Phonon::AudioOutput *audioOutput =
                new Phonon::AudioOutput(Phonon::MusicCategory, this);
    Phonon::createPath(mediaObject, audioOutput);

    // 关联媒体对象的 tick()信号来更新播放时间的显示
    connect(mediaObject, SIGNAL(tick(qint64)), this, SLOT(updateTime(qint64)));

    …… //这里省略了部分代码,用于创建界面

    mediaObject->setCurrentSource(Phonon::MediaSource("../myPlayer/music.mp3"));
}
```

该函数用于初始化播放器,包括设置主界面外观、初始化所有变量、创建媒体图等,添加该函数主要是为了简化构造函数。注意:为了缩减篇幅,这里省略了创建界面部件的相关代码,读者需要从网站上下载该部分源码,将该函数补充完善后再继续下面的内容。

在播放媒体文件时,媒体对象 MediaObject 会在指定的时间间隔发射 tick(qint64 time)信号,这个时间间隔可以使用 setTickInterval()来进行设置。tick()中的 time 参数指定了媒体对象在媒体流中的当前位置,单位是毫秒。通过关联这个信号就可以获取当前播放的时间。下面是更新播放时间的槽的定义:

```
// 更新 timeLabel 标签显示的播放时间
void MyWidget::updateTime(qint64 time)
{
    qint64 totalTimeValue = mediaObject->totalTime();
    QTime totalTime(0, (totalTimeValue / 60000) % 60, (totalTimeValue / 1000) % 60);
    QTime currentTime(0, (time / 60000) % 60, (time / 1000) % 60);
    QString str = currentTime.toString("mm:ss") + " / " + totalTime.toString("mm:ss");
    timeLabel->setText(str);
}
```

这里使用 MediaObject 类中的 totalTime()函数获取了当前播放歌曲的总时间,单位也是毫秒。然后通过转换将当前时间和总时间显示在 timeLabel 标签中。

下面是播放动作 playAction 的触发信号对应的槽的定义,用来实现歌曲的播放和暂停的控制:

```
// 播放或者暂停
void MyWidget::setPaused()
{
    // 如果先前处于播放状态,那么暂停播放;否则,开始播放
    if (mediaObject->state() == Phonon::PlayingState)
```

```
        mediaObject->pause();
    else
        mediaObject->play();
}
```

这里使用了 MediaObject 的 state()函数来获取媒体对象所处于的状态，这个会在下一小节详细讲解。最后再添加上其他几个槽的定义，因为它们的功能涉及了后面的内容，所以现在先不添加它们的实现代码：

```
// 播放上一首，与 skipBackwardAction 动作的触发信号关联
void MyWidget::skipBackward()
{
}
// 播放下一首，与 skipForwardAction 动作的触发信号关联
void MyWidget::skipForward()
{
}
// 打开文件，与 openAction 动作的触发信号关联
void MyWidget::openFile()
{
}
// 显示或者隐藏播放列表，与 PLAction 动作的触发信号关联
void MyWidget::setPlaylistShown()
{
}
// 显示或者隐藏桌面歌词，与 LRCAction 动作的触发信号关联
void MyWidget::setLrcShown()
{
}
```

因为在前面的代码中使用了中文，所以还要在 main()函数中添加相关代码。现在运行程序，就可以出现图 3-1 中的播放器主界面了，因为在 initPlayer()函数的最后指定了媒体源，所以现在可以按下播放图标来播放指定的歌曲。

### 3.2.2 媒体对象状态

在 Phonon 中将媒体文件的播放划分为了几个状态，媒体对象 MediaObject 总是处于这几种状态的其中一个。这几种状态由 Phonon::State 枚举变量来定义，其取值如表 3-1 所列。媒体对象可以在任意时间变化到任意的状态，而无论其先前处于什么状态，当媒体对象的状态发生改变时就会发射 stateChanged()信号，可以通过关联该信号来获取媒体对象当前的状态，从而进行一些有关的设置，例如改变播放图标的状态等。

表 3-1　媒体对象的各种状态

| 常　量 | 描　述 |
| --- | --- |
| Phonon::LoadingState | 加载状态。从构建完媒体对象直到它准备好进行播放前所处的状态。这个状态一般用于后端初始化媒体图和加载媒体源。当媒体对象离开加载状态时，就会进入 StoppedState，除非发生了错误，或者通过调用函数而要进入其他状态，比如调用了 play()函数，那么就会直接进入 PlayingState |
| Phonon::StoppedState | 停止状态。在该状态媒体对象已经准备好了进行当前媒体源的播放。处于该状态时媒体对象在媒体流中的位置为 0。调用 stop()函数可以直接进入该状态 |
| Phonon::PlayingState | 播放状态。媒体对象正在播放媒体源。调用 play()函数可以直接进入该状态 |
| Phonon::BufferingState | 缓冲状态。播放器正在等待数据以便开始播放或者继续播放。这个状态一般用于等待通过网络连接的媒体数据 |
| Phonon::PausedState | 暂停状态。播放器暂停当前的播放。调用 pause()函数可以直接进入暂停状态 |
| Phonon::ErrorState | 错误状态。当播放中出现问题时就会进入错误状态。错误类型由枚举变量 Phonon::ErrorType 来定义。如果是 FatalError，那么播放将无法继续，但是可以尝试使用新的媒体源；如果是 NormalError，则可以继续播放，不过媒体对象会改变状态到 ErrorState |

当创建了媒体对象后，它就会处于 LoadingState 状态，只有使用 createPath()为其设置了 Path，再使用 setCurrentSource()为其设置了当前媒体源以后，媒体对象才会进入 StoppedState 状态。当然，如果在设置了媒体源之后立即调用了 play()函数，那么媒体对象就不会进入 StoppedState 状态，而是直接进入 PlayingState 状态。

下面继续在程序中添加代码，使用媒体的各种状态来改变几个动作图标的状态。(项目源码路径：src\3\3-2\myPlayer)首先在 mywidget.h 文件中添加一个私有槽的声明：

```
void stateChanged(Phonon::State newState, Phonon::State oldState);
```

然后到 mywidget.cpp 文件中，先添加头文件：

```
#include <QMessageBox>
#include <QFileInfo>
```

再在 initPlayer()函数的最后添加如下一行代码：

```
connect(mediaObject, SIGNAL(stateChanged(Phonon::State, Phonon::State)),
```

```
        this, SLOT(stateChanged(Phonon::State, Phonon::State)));
```

下面添加stateChanged()槽的定义：

```
// 媒体对象状态发生了改变
void MyWidget::stateChanged(Phonon::State newState, Phonon::State oldState)
{
  switch (newState)
  {
  case Phonon::ErrorState :
    if(mediaObject->errorType() == Phonon::FatalError) {
      QMessageBox::warning(this, tr("致命错误"),
                           mediaObject->errorString());
    } else {
      QMessageBox::warning(this, tr("错误"), mediaObject->errorString());
    }
    break;
  case Phonon::PlayingState :
    stopAction->setEnabled(true);
    playAction->setIcon(QIcon(":/images/pause.png"));
    playAction->setText(tr("暂停(F5)"));
    topLabel->setText(QFileInfo(mediaObject->
                      currentSource().fileName()).baseName());
    break;
  case Phonon::StoppedState :
    stopAction->setEnabled(false);
    playAction->setIcon(QIcon(":/images/play.png"));
    playAction->setText(tr("播放(F5)"));
    topLabel->setText(tr("<a href = \" http://"www.yafeilinux.com \">
                      www.yafeilinux.com</a>"));
    timeLabel->setText(tr("00:00 / 00:00"));
    break;
  case Phonon::PausedState :
    stopAction->setEnabled(true);
    playAction->setIcon(QIcon(":/images/play.png"));
    playAction->setText(tr("播放(F5)"));
    topLabel->setText(QFileInfo(mediaObject->
                currentSource().fileName()).baseName() + tr(" 已暂停!"));
    break;
  case Phonon::BufferingState :
    break;
  default :
    ;
  }
}
```

这里就是根据媒体对象的不同状态进行了不同的处理。其中 currentSource().fileName()可以获取当前播放的媒体源的路径，然后使用 QFileInfo 类的 baseName()函数可以获取其中的文件名，QFileInfo 类在《Qt Creator 快速入门》中第 15 章讲到。现在运行程序，使用播放和停止图标来控制播放，可以看到它们状态的变化。

## 3.3　实现播放列表

要实现播放列表功能，先要设计一个播放列表类，然后再进行歌曲信息的解析，使用歌曲标题、艺术家和长度等信息来创建播放列表。

### 3.3.1　设计播放列表

设计播放列表主要是设计 MyPlaylist 类，继承自 QTableWidget 类。这里需要实现两个功能：在播放列表上右击弹出菜单来清空播放列表；关闭播放列表时只是隐藏它而不让它退出。

（项目源码路径：src\3\3 - 3\myPlayer）首先向项目中添加新文件，模板选择 C++类，类名为 MyPlaylist，基类为 QTableWidget，类型信息选择“继承自 QWidget”。完成后在 myplaylist.h 文件中添加一些函数、信号、槽的声明：

```
protected:
  void contextMenuEvent(QContextMenuEvent * event);
  void closeEvent(QCloseEvent * event);
signals:
  void playlistClean();
private slots:
  void clearPlaylist();
```

然后到 myplaylist.cpp 文件中，先添加头文件：

```
#include <QContextMenuEvent>
#include <QMenu>
```

再在构造函数中初始化播放列表的相关设置：

```
setWindowTitle(tr("播放列表"));

// 设置窗口标志，表明它是一个独立窗口且有一个只带有关闭按钮的标题栏
setWindowFlags(Qt::Window | Qt::WindowTitleHint);

// 设置初始大小，并且锁定部件宽度
resize(320, 400);
setMaximumWidth(320);
setMinimumWidth(320);
```

```
    // 设置行列数目
    setRowCount(0);
    setColumnCount(3);

    // 设置表头标签
    QStringList list;
    list << tr("标题") << tr("艺术家") << tr("长度");
    setHorizontalHeaderLabels(list);

    // 设置只能选择单行
    setSelectionMode(QAbstractItemView::SingleSelection);
    setSelectionBehavior(QAbstractItemView::SelectRows);

    // 设置不显示网格
    setShowGrid(false);
```

然后是几个函数和槽的定义：

```
// 上下文菜单事件处理函数，当点击鼠标右键时运行一个菜单
void MyPlaylist::contextMenuEvent(QContextMenuEvent * event)
{
    QMenu menu;
    menu.addAction(tr("清空列表"), this, SLOT(clearPlaylist()));
    menu.exec(event->globalPos());
}
// 清空播放列表
void MyPlaylist::clearPlaylist()
{
    while (rowCount())
        removeRow(0);

    // 发射播放列表已清空信号
    emit playlistClean();
}
// 关闭事件处理函数，如果部件处于显示状态，则使其隐藏
void MyPlaylist::closeEvent(QCloseEvent * event)
{
    if (isVisible()) {
        hide();
        event->ignore();
    }
}
```

在清空播放列表时发射了信号，这个信号后面会在 MyWidget 类中用到。

### 3.3.2　创建播放列表

这里要使用歌曲的标题、艺术家和长度等信息来创建播放列表。可以使用 MediaObject 的 metaData()函数获取媒体源中的元数据(meta data),它是个键值对,其中包含的信息如表 3-2 所列。这样就可以通过解析元数据来获取歌曲的相关信息。

**表 3-2　媒体元数据键值**

| 键 | 描　述 |
|---|---|
| ARTIST | 艺术家 |
| ALBUM | 专辑 |
| TITLE | 标题 |
| DATE | 日期 |
| GENRE | 风格 |
| TRACKNUMBER | 音轨编号 |
| DESCRIPTION | 内容描述 |
| MUSICBRAINZDISCID | MusicBrainz 光盘标识 |

为了不产生混乱,程序没有使用用于歌曲播放的 mediaObject 对象,而是创建了新的 MediaObject 类对象 metaInformationResolver 作为元数据的解析器。因为只有在 LoadingState 完成后才能获取元数据,所以可以先调用解析器的 setCurrentSource()函数为其设置一个媒体源,然后关联它的 stateChanged()信号,等其进入到 StoppedState 状态后再进行元数据的解析。程序还使用了一个 MediaSource 类型的 QList 列表 sources 来保存所有的媒体源,当清空播放列表时,也要清空 sources,这就需要使用在 MyPlaylist 类中设计的 playlistClean()信号。

另外,只是使用前一节中讲到的媒体对象状态来设置播放、停止等动作图标的状态是不够的,还要根据播放列表的状态来更新这些图标的状态,比如,当播放列表被清空且播放器当前没有处于播放状态时,所有的图标都要处于不可用状态。还有就是,当一个歌曲将要播放完毕时,还要判断在播放列表中,该歌曲的后面是否还有其他歌曲,如果有,则要使用 enqueue()将其添加到媒体对象的播放队列中,这样它就可以自动播放了。

继续在前面的程序中添加代码。首先到 mywidget.h 文件中添加类的前置声明:

```
class MyPlaylist;
```

然后添加私有槽声明:

```
void sourceChanged(const Phonon::MediaSource &source);
```

```
void aboutToFinish();
void metaStateChanged(Phonon::State newState, Phonon::State oldState);
void tableClicked(int row);
void clearSources();
```

再添加几个私有变量的定义和函数声明：

```
MyPlaylist * playlist;
Phonon::MediaObject * metaInformationResolver;
QList<Phonon::MediaSource> sources;
void changeActionState();
```

下面到 mywidget.cpp 文件中，先添加头文件：

```
#include "myplaylist.h"
#include <QFileDialog>
#include <QDesktopServices>
```

然后到 initPlayer()函数中，先删除以前那行用来设置当前媒体源的代码：

```
mediaObject->setCurrentSource(Phonon::MediaSource("../myPlayer/music.mp3"));
```

然后添加如下代码：

```
// 创建播放列表
playlist = new MyPlaylist(this);
connect(playlist, SIGNAL(cellClicked(int, int)), this, SLOT(tableClicked(int)));
connect(playlist, SIGNAL(playlistClean()), this, SLOT(clearSources()));

// 创建用来解析媒体的信息的元信息解析器
metaInformationResolver = new Phonon::MediaObject(this);

// 需要与 AudioOutput 连接后才能使用 metaInformationResolver 来获取歌曲的总时间
Phonon::AudioOutput * metaInformationAudioOutput =
                new Phonon::AudioOutput(Phonon::MusicCategory, this);
Phonon::createPath(metaInformationResolver, metaInformationAudioOutput);
connect(metaInformationResolver,
    SIGNAL(stateChanged(Phonon::State,Phonon::State)),this,
    SLOT(metaStateChanged(Phonon::State, Phonon::State)));

connect(mediaObject, SIGNAL(currentSourceChanged(Phonon::MediaSource)),
    this, SLOT(sourceChanged(Phonon::MediaSource)));
connect(mediaObject, SIGNAL(aboutToFinish()), this, SLOT(aboutToFinish()));

// 初始化动作图标的状态
playAction->setEnabled(false);
stopAction->setEnabled(false);
skipBackwardAction->setEnabled(false);
```

```
skipForwardAction->setEnabled(false);
topLabel->setFocus();
```

现在已经创建了播放列表对象，可以使用界面上的 PL 图标来显示播放列表窗口了。下面更改 setPlaylistShown()槽的定义：

```
void MyWidget::setPlaylistShown()
{
  if (playlist->isHidden()) {
    playlist->move(frameGeometry().bottomLeft());
    playlist->show();
  } else {
    playlist->hide();
  }
}
```

如果播放列表处于隐藏状态，那么就让它显示在主界面的下方；如果播放列表处于显示状态，则隐藏它。我们要将打开的文件添加到播放列表中，所以先将打开文件 openFile()槽的定义更改如下：

```
void MyWidget::openFile()
{
  // 从系统音乐目录打开多个音乐文件
  QStringList list = QFileDialog::getOpenFileNames(this, tr("打开音乐文件"),
    QDesktopServices::storageLocation(QDesktopServices::MusicLocation));
  if (list.isEmpty())
    return;

  // 获取当前媒体源列表的大小
  int index = sources.size();

  // 将打开的音乐文件添加到媒体源列表后
  foreach (QString string, list) {
    Phonon::MediaSource source(string);
    sources.append(source);
  }
 // 如果媒体源列表不为空，则将新加入的第一个媒体源作为当前媒体源，
 // 这时会发射 stateChanged()信号，从而调用 metaStateChanged()函数进行媒体源的解析
  if (! sources.isEmpty())
    metaInformationResolver->setCurrentSource(sources.at(index));
}
```

打开歌曲文件后，将它们设置为媒体源添加到媒体源列表的后面，然后调用函数 setCurrentSource()设置新添加的第一个媒体源为解析器的当前媒体源，这时解析器就会改变状态而进入加载状态，等加载完成后，又会进入停止状态。因为在 initPlay-

er()函数中关联了解析器状态改变时发射的stateChanged()信号到metaStateChanged()槽上，所以这时就会执行该槽来进行媒体源的解析。下面添加metaStateChanged()槽的定义：

```
// 解析媒体文件的元信息
void MyWidget::metaStateChanged(Phonon::State newState, Phonon::State oldState)
{
  // 错误状态，则从媒体源列表中除去新添加的媒体源
  if(newState == Phonon::ErrorState) {
    QMessageBox::warning(this, tr("打开文件时出错"), metaInformationResolver->
errorString());
    while (! sources.isEmpty() && ! (sources.takeLast() ==
        metaInformationResolver->currentSource()))
    {};
    return;
  }
  // 如果既不处于停止状态也不处于暂停状态，则直接返回
  if(newState != Phonon::StoppedState && newState != Phonon::PausedState)
    return;

  // 如果媒体源类型错误，则直接返回
  if(metaInformationResolver->currentSource().type()
    == Phonon::MediaSource::Invalid)
    return;

  // 获取媒体信息
  QMap<QString, QString> metaData = metaInformationResolver->metaData();

  // 获取标题，如果为空，则使用文件名
  QString title = metaData.value("TITLE");
  if (title == "") {
    QString str = metaInformationResolver->currentSource().fileName();
    title = QFileInfo(str).baseName();
  }
  QTableWidgetItem *titleItem = new QTableWidgetItem(title);
  // 设置数据项不可编辑
  titleItem->setFlags(titleItem->flags() ^ Qt::ItemIsEditable);

  // 获取艺术家信息
  QTableWidgetItem *artistItem =
                    new QTableWidgetItem(metaData.value("ARTIST"));
  artistItem->setFlags(artistItem->flags() ^ Qt::ItemIsEditable);

  // 获取总时间信息
  qint64 totalTime = metaInformationResolver->totalTime();
```

```
    QTime time(0, (totalTime / 60000) % 60, (totalTime / 1000) % 60);
    QTableWidgetItem * timeItem = new QTableWidgetItem(time.toString("mm:ss"));

    // 插入到播放列表
    int currentRow = playlist->rowCount();
    playlist->insertRow(currentRow);
    playlist->setItem(currentRow, 0, titleItem);
    playlist->setItem(currentRow, 1, artistItem);
    playlist->setItem(currentRow, 2, timeItem);

    // 如果添加的媒体源还没有解析完,那么继续解析下一个媒体源
    int index = sources.indexOf(metaInformationResolver->currentSource()) + 1;
    if (sources.size() > index) {
      metaInformationResolver->setCurrentSource(sources.at(index));
    } else { // 如果所有媒体源都已经解析完成
      // 如果播放列表中没有选中的行
      if (playlist->selectedItems().isEmpty()) {
        // 如果现在没有播放歌曲则设置第一个媒体源为媒体对象的当前媒体源
        //(因为可能正在播放歌曲时清空了播放列表,然后又添加了新的列表)
        if (mediaObject->state() != Phonon::PlayingState &&
            mediaObject->state() != Phonon::PausedState) {
          mediaObject->setCurrentSource(sources.at(0));
        } else {
          //如果正在播放歌曲,则选中播放列表的第一个曲目,并更改图标状态
          playlist->selectRow(0);
          changeActionState();
        }
      else {
        // 如果播放列表中有选中的行,那么直接更新图标状态
        changeActionState();
      }
    }
  }
```

这个槽主要完成了媒体源的元数据的解析功能,并将解析出来的信息添加到了播放列表中。当播放列表的内容改变时,还要更改主界面上图标的状态,这个由changeActionState()函数来完成,下面添加该函数的定义:

```
// 根据媒体源列表内容和当前媒体源的位置来改变主界面图标的状态
void MyWidget::changeActionState()
{
  // 如果媒体源列表为空
  if (sources.count() == 0) {
    // 如果没有在播放歌曲,则播放和停止按钮都不可用
    //(因为可能歌曲正在播放时清除了播放列表)
```

```
        if (mediaObject->state() != Phonon::PlayingState
            && mediaObject->state() != Phonon::PausedState) {
            playAction->setEnabled(false);
            stopAction->setEnabled(false);
        }
        skipBackwardAction->setEnabled(false);
        skipForwardAction->setEnabled(false);
    } else { // 如果媒体源列表不为空
        playAction->setEnabled(true);
        stopAction->setEnabled(true);
        // 如果媒体源列表只有一行
        if (sources.count() == 1) {
            skipBackwardAction->setEnabled(false);
            skipForwardAction->setEnabled(false);
        } else { // 如果媒体源列表有多行
            skipBackwardAction->setEnabled(true);
            skipForwardAction->setEnabled(true);
            int index = playlist->currentRow();
            // 如果播放列表当前选中的行为第一行
            if (index == 0)
                skipBackwardAction->setEnabled(false);
            // 如果播放列表当前选中的行为最后一行
            if (index + 1 == sources.count())
                skipForwardAction->setEnabled(false);
        }
    }
}
```

当播放的媒体源改变时，媒体对象会发射currentSourceChanged()信号，这时要在播放列表中选中对应的曲目，然后更新图标的状态。下面添加sourceChanged()槽的定义：

```
// 当媒体源改变时，在播放列表中选中相应的行并更新图标的状态
void MyWidget::sourceChanged(const Phonon::MediaSource &source)
{
    int index = sources.indexOf(source);
    playlist->selectRow(index);
    changeActionState();
}
```

当播放器中当前的歌曲将要播放完毕时，媒体对象会发射aboutToFinish()信号，我们需要判断媒体源列表中是否还有后续的歌曲，如果有就要将其添加到播放队列中。下面添加aboutToFinish()槽的定义：

```
// 当前媒体源播放将要结束时，如果在列表中当前媒体源的后面还有媒体源
```

```
// 那么将它添加到播放队列中,否则停止播放
void MyWidget::aboutToFinish()
{
  int index = sources.indexOf(mediaObject->currentSource()) + 1;
  if (sources.size() > index) {
    mediaObject->enqueue(sources.at(index));

    // 跳转到歌曲最后
    mediaObject->seek(mediaObject->totalTime());
  } else {
    mediaObject->stop();
  }
}
```

媒体对象队列中的媒体源会一个接一个地自动播放,而且在两个媒体源切换时不会改变媒体对象的状态,也就是说,在当前媒体源播放时,媒体对象处于 PlayingState 状态,那么当前媒体源播放结束后继续播放队列中的下一个媒体源时,媒体对象不会发生任何状态变化,依然处于 PlayingState 状态。如果想在播放结束前一个指定的时间发射信号,那么可以使用 prefinishMarkReached()信号,使用 setPrefinishMark()来设置时间。

当在播放列表中单击了一首歌曲时,要将其设置为当前的媒体源,并根据媒体对象的状态来判断是否进行播放。下面添加 tableClicked()槽的定义:

```
// 单击播放列表
void MyWidget::tableClicked(int row)
{
  // 首先获取媒体对象当前的状态,然后停止播放并清空播放队列
  bool wasPlaying = mediaObject->state() == Phonon::PlayingState;
  mediaObject->stop();
  mediaObject->clearQueue();

  // 如果单击的播放列表中的行号大于媒体源列表的大小,则直接返回
  if(row >= sources.size())
    return;

  // 设置单击的行对应的媒体源为媒体对象的当前媒体源
  mediaObject->setCurrentSource(sources.at(row));

  // 如果以前媒体对象处于播放状态,那么开始播放选中的曲目
  if (wasPlaying)
    mediaObject->play();
}
```

当清空了播放列表时也要清空媒体源列表。下面添加 clearSources()槽的定义:

```
// 清空媒体源列表,它与播放列表的 playListClean()信号关联
void MyWidget::clearSources()
{
  sources.clear();

  // 更改动作图标状态
  changeActionState();
}
```

最后再更改 skipBackward()和 skipForward()的定义如下:

```
void MyWidget::skipBackward()
{
  int index = sources.indexOf(mediaObject->currentSource());
  mediaObject->setCurrentSource(sources.at(index - 1));
  mediaObject->play();
}
void MyWidget::skipForward()
{
  int index = sources.indexOf(mediaObject->currentSource());
  mediaObject->setCurrentSource(sources.at(index + 1));
  mediaObject->play();
}
```

此时运行程序,单击 PL 图标就会出现图 3-1 中所示的播放列表窗口,可以打开一些歌曲文件进行各种播放控制。

## 3.4 实现桌面歌词

要实现歌词的显示,先要了解歌词文件的格式和内容。现在歌曲文件主要使用的是 LRC(lyric 的缩写)歌词,该歌词以.lrc 为后缀,只包含纯文本的内容,可以使用任意一个文本编辑器将其打开。下面是一个标准的 LRC 歌词的部分内容:

```
[ti:我是明星]
[ar:周华健]
[al:奥运志愿者招募歌曲]
[by:yafeilinux]
[00:01.65]我是明星
[00:05.97]周华健
[00:10.26]LRC 制作:www.yafeilinux.com
[01:10.79][00:14.65]有一个梦,由我启动,
[01:15.26][00:19.02]把汗水融化成满脸笑容。
[01:19.71][00:3.47]海阔天空,我是阵风,
[01:24.20][00:28.07]把旗帜飞扬到南北西东。
[01:28.66][00:32.62]嘿呀,嘿呀。谁不为人心的光辉感动。
```

```
[01:37.75][00:41.48]嘿呀,嘿呀。我的心就是个光明火种。
[02:00.27][00:52.72]每一个人,一样有用。
[02:04.74][00:57.17]自告奋勇,不约而同。
[02:09.29][01:01.66]忘了自己,快乐心胸。
[02:13.67][01:06.18]我是明星,点缀天空。
[01:52.44][02:54.31]
[02:18.27]有一个梦,由我启动,
[02:22.72]等待着你发自内心笑容。
```

其中,[标识名:值]是标志标签,一般有以下几种标志标签:[ar:歌手]、[ti:歌曲标题]、[al:专辑名]和[by:编辑者]。而[00:01.65]是时间标签,其中 00 表示分,01 表示秒,65 表示十毫秒,就是说在播放到 00 分 01 秒 650 毫秒的时候显示歌词“我是明星”。时间标签可以处于歌词中的任何位置,一般是处于最左边。也可能在一首歌曲中要多次重复出现相同的歌词,这时在一个歌词中可能会出现多个时间标签。

了解了 LRC 歌词的文件格式,那么就要想办法对其进行解析。因为一般只需要解析时间标签和对应的歌词,而不用解析其他的标志标签,所以应该联想到 QMap 容器。可以使用 QMap 来存储<时间,歌词>对,而且 QMap 是按照键值顺序来存储的,也就是说它可以自动按照时间将歌词进行排序,这将大大缩短解析的时间。

而桌面歌词就是将歌词显示在桌面上,这个可以使用背景透明的 QLabel 标签部件来实现。

### 3.4.1　设计桌面歌词部件

要实现桌面歌词主要是设计 MyLrc 类,它继承自 QLabel 类。这里需要将该部件的背景设置为透明,然后重新实现其重绘事件处理函数来自定义文本的显示,这里可以使用渐变填充来实现多彩的文字。然后再使用定时器,在已经绘制的歌词上面再绘制一个不断变宽的相同的歌词来实现动态遮罩效果。

(项目源码路径:src\3\3-4\myPlayer)首先向项目中添加新的 C++类,类名为 MyLrc,基类为 QLabel,类型信息选择“继承自 QWidget”。完成后将 mylrc.h 文件的内容更改如下:

```
#include <QLabel>
class QTimer;

class MyLrc : public QLabel
{
  Q_OBJECT
public:
  explicit MyLrc(QWidget *parent = 0);
  void startLrcMask(qint64 intervalTime);
  void stopLrcMask();
```

```
protected:
    void paintEvent(QPaintEvent * );
    void mousePressEvent(QMouseEvent * event);
    void mouseMoveEvent(QMouseEvent * event);
    void contextMenuEvent(QContextMenuEvent * event);

private slots:
    void timeout();

private:
    QLinearGradient linearGradient;
    QLinearGradient maskLinearGradient;
    QFont font;
    QTimer * timer;
    qreal lrcMaskWidth;

    // 每次歌词遮罩增加的宽度
    qreal lrcMaskWidthInterval;
    QPoint offset;
};
```

然后到 mylrc.cpp 文件中,先添加头文件:

```
#include <QPainter>
#include <QTimer>
#include <QMouseEvent>
#include <QContextMenuEvent>
#include <QMenu>
```

然后在构造函数中添加如下代码:

```
setWindowFlags(Qt::Window | Qt::FramelessWindowHint);

// 设置背景透明
setAttribute(Qt::WA_TranslucentBackground);
setText(tr("MyPlayer 音乐播放器 - - - yafeilinux 作品"));

// 固定部件大小
setMaximumSize(800, 60);
setMinimumSize(800, 60);

// 歌词的线性渐变填充
linearGradient.setStart(0, 10);
linearGradient.setFinalStop(0, 40);
linearGradient.setColorAt(0.1, QColor(14, 179, 255));
linearGradient.setColorAt(0.5, QColor(114, 32, 255));
```

```
linearGradient.setColorAt(0.9, QColor(14, 179, 255));

// 遮罩的线性渐变填充
maskLinearGradient.setStart(0, 10);
maskLinearGradient.setFinalStop(0, 40);
maskLinearGradient.setColorAt(0.1, QColor(222, 54, 4));
maskLinearGradient.setColorAt(0.5, QColor(255, 72, 16));
maskLinearGradient.setColorAt(0.9, QColor(222, 54, 4));
// 设置字体
font.setFamily("Times New Roman");
font.setBold(true);
font.setPointSize(30);

// 设置定时器
timer = new QTimer(this);
connect(timer, SIGNAL(timeout()), this, SLOT(timeout()));
lrcMaskWidth = 0;
lrcMaskWidthInterval = 0;
```

下面是重绘事件处理函数的定义：

```
void MyLrc::paintEvent(QPaintEvent *)
{
  QPainter painter(this);
  painter.setFont(font);

  // 先绘制底层文字,作为阴影,这样会使显示效果更加清晰,且更有质感
  painter.setPen(QColor(0, 0, 0, 200));
  painter.drawText(1, 1, 800, 60, Qt::AlignLeft, text());

  // 再在上面绘制渐变文字
  painter.setPen(QPen(linearGradient, 0));
  painter.drawText(0, 0, 800, 60, Qt::AlignLeft, text());

  // 设置歌词遮罩
  painter.setPen(QPen(maskLinearGradient, 0));
  painter.drawText(0, 0, lrcMaskWidth, 60, Qt::AlignLeft, text());
}
```

这里一共将相同的歌词绘制了 3 次：第 1 次绘制了深黑色的歌词，它最先绘制，所以处于最底层，而且起始位置为(1,1)，所以不会和上面的歌词完全重合，用来实现阴影效果；第 2 次绘制了渐变填充的歌词，它是要正常显示的歌词；第 3 次绘制的是用于实现遮罩的歌词，它的颜色与前两次不同，且宽度是变化的，由歌词的长度和定时器来决定它的宽度，这样就实现了在播放时歌词从左向右动态改变颜色的效果。

下面添加其他一些槽和函数的定义：

```
// 开启遮罩,需要指定当前歌词开始与结束之间的时间间隔
void MyLrc::startLrcMask(qint64 intervalTime)
{
  // 这里设置每隔 30 毫秒更新一次遮罩的宽度,因为如果更新太频繁
  // 会增加 CPU 占用率,而如果时间间隔太大,则动画效果就不流畅了
  qreal count = intervalTime / 30;

  // 获取遮罩每次需要增加的宽度,这里的 800 是部件的固定宽度
  lrcMaskWidthInterval = 800 / count;
  lrcMaskWidth = 0;
  timer->start(30);
}
// 停止遮罩
void MyLrc::stopLrcMask()
{
  timer->stop();
  lrcMaskWidth = 0;
  update();
}
// 两个鼠标事件处理函数实现了部件的拖动
void MyLrc::mousePressEvent(QMouseEvent * event)
{
  if (event->button() == Qt::LeftButton)
    offset = event->globalPos() - frameGeometry().topLeft();
}
void MyLrc::mouseMoveEvent(QMouseEvent * event)
{
  if (event->buttons() & Qt::LeftButton) {
  setCursor(Qt::PointingHandCursor);
  move(event->globalPos() - offset);
  }
}
// 实现右键菜单来隐藏部件
void MyLrc::contextMenuEvent(QContextMenuEvent * event)
{
  QMenu menu;
  menu.addAction(tr("隐藏"), this, SLOT(hide()));
  menu.exec(event->globalPos());
}
// 定时器溢出时增加遮罩的宽度,并更新显示
void MyLrc::timeout()
{
  lrcMaskWidth += lrcMaskWidthInterval;
  update();
}
```

这些函数都是用来辅助实现桌面歌词的，因为都很简单，所以不再逐一介绍。

### 3.4.2　解析歌词文件

歌词文件的解析由 MyWidget 类的 resolveLrc()函数来实现。一般的，歌词文件会和歌曲文件在同一目录中，且与歌曲文件同名，这里就是通过把歌曲文件的后缀改为".lrc"来获取歌词文件路径的。然后使用了正则表达式来获取歌词文件中的时间和歌词文本，并将它们存储到了 QMap<qint64, QString>类型的对象 lrcMap 中，其中键值是以毫秒为单位的数值，而对应的值是相应的歌词文本。当获取了歌词信息后，需要适时的将它们显示出来，这个可以在 updateTime()槽中来完成，每当更新时间时都同步更新歌词的显示。

下面继续在前面的程序中添加代码。先到 mywidget.h 文件中添加类的前置声明：

```
class MyLrc;
```

然后添加私有对象的定义和函数的声明：

```
MyLrc * lrc;
QMap<qint64, QString> lrcMap;
void resolveLrc(const QString &sourceFileName);
```

再到 mywidget.cpp 文件中，先添加头文件：

```
#include "mylrc.h"
#include <QTextCodec>
```

再到 initPlayer()函数的最后添加歌词部件的创建代码：

```
lrc = new MyLrc(this);
```

因为现在已经创建了歌词部件，所以可以使用 LRC 动作图标来显示桌面歌词了，更改 setLrcShown()槽的定义如下：

```
void MyWidget::setLrcShown()
{
  if (lrc->isHidden())
    lrc->show();
  else
    lrc->hide();
}
```

下面添加歌词的解析函数的定义，它需要传入歌曲文件的路径：

```
// 解析 LRC 歌词
void MyWidget::resolveLrc(const QString &sourceFileName)
{
```

```
// 先清空以前的内容
lrcMap.clear();

// 获取 LRC 歌词的文件名
if(sourceFileName.isEmpty())
  return ;
QString fileName = sourceFileName;
QString lrcFileName = fileName.remove(fileName.right(3)) + "lrc";

// 打开歌词文件
QFile file(lrcFileName);
if (! file.open(QIODevice::ReadOnly)) {
  lrc->setText(QFileInfo(mediaObject->currentSource()
      .fileName()).baseName() + tr(" - - - 未找到歌词文件!"));
  return ;
}
// 设置字符串编码
QTextCodec::setCodecForCStrings(QTextCodec::codecForLocale());
// 获取全部歌词信息
QString allText = QString(file.readAll());

// 关闭歌词文件
file.close();

// 将歌词按行分解为歌词列表
QStringList lines = allText.split("\n");

// 使用正则表达式将时间标签和歌词内容分离
QRegExp rx("\\[\\d{2}:\\d{2}\\.\\d{2}\\]");
foreach (QString oneLine, lines) {
  // 先在当前行的歌词的备份中将时间内容清除,这样就获得了歌词文本
  QString temp = oneLine;
  temp.replace(rx, "");
  // 然后依次获取当前行中所有时间标签,并分别与歌词文本存入 QMap 中
  int pos = rx.indexIn(oneLine, 0);
  while (pos != -1) {
    QString cap = rx.cap(0);

    // 将时间标签转换为时间数值,以毫秒为单位
    QRegExp regexp;
    regexp.setPattern("\\d{2}(?=:)");
    regexp.indexIn(cap);
    int minute = regexp.cap(0).toInt();
    regexp.setPattern("\\d{2}(?=\\.)");
    regexp.indexIn(cap);
```

```
            int second = regexp.cap(0).toInt();
            regexp.setPattern("\\d{2}(? =\\])");
            regexp.indexIn(cap);
            int millisecond = regexp.cap(0).toInt();
            qint64 totalTime = minute * 60000 + second * 1000 + millisecond * 10;

            // 插入到 lrcMap 中
            lrcMap.insert(totalTime, temp);
            pos += rx.matchedLength();
            pos = rx.indexIn(oneLine, pos);
        }
    }
    // 如果 lrcMap 为空
    if (lrcMap.isEmpty()) {
        lrc->setText(QFileInfo(mediaObject->currentSource()
            .fileName()).baseName() + tr(" - - - 歌词文件内容错误!"));
    return;
    }
}
```

为了可以在播放时更改歌词内容的显示，在 updateTime()函数的最后添加如下代码：

```
// 获取当前时间对应的歌词
if (! lrcMap.isEmpty()) {
    // 获取当前时间在歌词中的前后两个时间点
    qint64 previous = 0;
    qint64 later = 0;
    foreach (qint64 value, lrcMap.keys()) {
        if (time >= value) {
            previous = value;
        } else {
            later = value;
            break;
        }
    }
    // 达到最后一行，将 later 设置为歌曲总时间的值
    if (later == 0)
        later = totalTimeValue;

    // 获取当前时间所对应的歌词内容
    QString currentLrc = lrcMap.value(previous);

    // 没有内容时
    if (currentLrc.length() < 2)
```

```
        currentLrc = tr("MyPlayer音乐播放器 - - - yafeilinux作品");

        // 如果是新的一行歌词,那么重新开始显示歌词遮罩
        if (currentLrc != lrc->text()) {
          lrc->setText(currentLrc);
          topLabel->setText(currentLrc);
          qint64 intervalTime = later - previous;
          lrc->startLrcMask(intervalTime);
        }
      } else { // 如果没有歌词文件,则在顶部标签中显示歌曲标题
        topLabel->setText(QFileInfo(mediaObject->
                        currentSource().fileName()).baseName());
      }
```

我们希望每次在播放歌曲时才进行歌词的解析,这样就可以在PlayingState状态来解析歌词。在stateChanged()函数的Phonon::PlayingState状态中的break代码前添加一行代码:

```
resolveLrc(mediaObject->currentSource().fileName());
```

这样就实现了在每次播放歌曲时自动解析其对应的歌词文件。因为在aboutToFinish()函数中设置播放队列中的歌曲的改变是不会进行状态转换的,所以还要在该函数中重新解析歌词。在seek(mediaObject->totalTime())函数调用的代码后添加如下代码:

```
lrc->stopLrcMask();
resolveLrc(sources.at(index).fileName());
```

为了使歌词显示更加人性化,还要在一些函数中添加代码。在skipBackward()函数和skipForward()函数的最开始,添加一行代码:

```
lrc->stopLrcMask();
```

这样每当跳转到其他歌曲时都会先停止歌词遮罩。下面再到stateChanged()槽的Phonon::Stopped状态中break代码前添加如下代码:

```
lrc->stopLrcMask();
lrc->setText(tr("MyPlayer音乐播放器 - - - yafeilinux作品"));
```

当歌曲停止播放时,停止歌词遮罩,并改变桌面歌词的显示文本。然后在Phonon::PausedState状态的break代码前添加如下代码:

```
// 如果该歌曲有歌词文件
if (! lrcMap.isEmpty()) {
  lrc->stopLrcMask();
  lrc->setText(topLabel->text());
}
```

如果暂停了播放，而且当前播放的歌曲正在显示歌词，那么就停止歌词遮罩，然后将桌面歌词的显示文本设置为顶部标签中的文本。

现在可以运行程序实现图 3-1 中所示的桌面歌词的显示效果。

## 3.5　添加系统托盘图标

对于一个完善的音乐播放器，大多数都有一个系统托盘图标，这样就可以在播放歌曲时将播放器主界面最小化到托盘图标了。Qt 中提供了 QSystemTrayIcon 类来实现系统托盘图标，可以很容易地为其添加菜单、设置工具提示、显示消息和处理各种交互。

(项目源码路径：src\3\3-5\myPlayer)首先在 widget.h 文件中添加头文件：

```
#include <QSystemTrayIcon>
```

然后添加一个私有槽声明：

```
void trayIconActivated(QSystemTrayIcon::ActivationReason activationReason);
```

再添加关闭事件处理函数的声明：

```
protected:
  void closeEvent(QCloseEvent * event);
```

最后添加私有对象定义：

```
QSystemTrayIcon * trayIcon;
```

然后进入 mywidget.cpp 文件，首先添加头文件：

```
#include <QMenu>
#include <QCloseEvent>
```

然后在 initPlayer()函数中构建系统托盘图标对象，并为其添加菜单：

```
// 创建系统托盘图标
trayIcon = new QSystemTrayIcon(QIcon(":/images/icon.png"), this);
trayIcon->setToolTip(tr("MyPlayer 音乐播放器 - - - yafeilinux 作品"));
// 创建菜单
QMenu * menu = new QMenu;
QList<QAction * > actions;
actions << playAction << stopAction << skipBackwardAction << skipForwardAc-
tion;
menu->addActions(actions);
menu->addSeparator();
menu->addAction(PLAction);
menu->addAction(LRCAction);
menu->addSeparator();
```

```
menu->addAction(tr("退出"), qApp, SLOT(quit()));
trayIcon->setContextMenu(menu);

// 托盘图标被激活后进行处理
connect(trayIcon, SIGNAL(activated(QSystemTrayIcon::ActivationReason)),
    this, SLOT(trayIconActivated(QSystemTrayIcon::ActivationReason)));

// 显示托盘图标
trayIcon->show();
```

最后添加槽和函数的定义：

```
// 系统托盘图标被激活
void MyWidget::trayIconActivated(QSystemTrayIcon::ActivationReason activationReason)
{
  // 如果单击了系统托盘图标,则显示应用程序界面
  if (activationReason == QSystemTrayIcon::Trigger) {
    show();
  }
}
// 关闭事件处理函数
void MyWidget::closeEvent(QCloseEvent * event)
{
  if (isVisible()) {
    hide();
  trayIcon->showMessage(tr("MyPlayer音乐播放器"), tr("点我重新显示主界面"));
    event->ignore();
  }
}
```

在关闭事件处理函数中,最后需要使用ignore()函数来忽略该事件,这样应用程序才不会退出。现在运行程序,已经可以实现图3-1中所示的播放器运行效果了。

到这里,整个音乐播放器程序就设计完成了,它实现了基本的音乐播放与控制功能,可以显示播放列表,还实现了桌面歌词功能。不过,因为篇幅有限,我们没有添加拖放、视觉效果(可以通过频谱分析来实现)、均衡器、界面换肤、迷你模式等现在很多音乐播放器都有的功能,这些功能读者可以通过前面学习的各章节知识自己实现,也可以参考一下我们网站上的音乐播放器的例子。

## 3.6 小 结

通过本章的学习,读者应该进一步掌握Phonon多媒体框架的应用,可以使用它实现影音播放的各种控制功能。在学习该播放器例子的同时,大家也应该加强自己综合运用知识的能力,因为很多应用程序都是要综合使用多方面的知识来实现的。

# 第 4 章　数据管理系统

这一章将讲述一个数据管理系统的实例，主要是应用《Qt Creator 快速入门》中数据应用篇相关章节的知识。该实例包括了密码登录、数据库操作、XML 文档操作和自定义视图等多个知识块的应用。因为任何一个实用的数据管理程序都会涉及复杂的逻辑和众多类型的数据，而这里重在讲解知识的应用，所以只是用了很简单地数据演示出来，希望读者在学习本章时可以将重点放在数据处理方法上。

该实例命名为数据管理系统，主要讲解了数据库的使用、XML 文档的操作以及自定义视图 3 部分内容，建议先学完《Qt Creator 快速入门》中第 17 章的内容再学本章。程序中我们使用了家电销售的一些数据作为例子，这些数据仅用作讲解知识，并不一定符合现实情况。为了控制程序规模，有些代码重复的功能在该程序中没有进行实现，可以自己添加。数据管理系统的最终运行界面如图 4-1 所示。

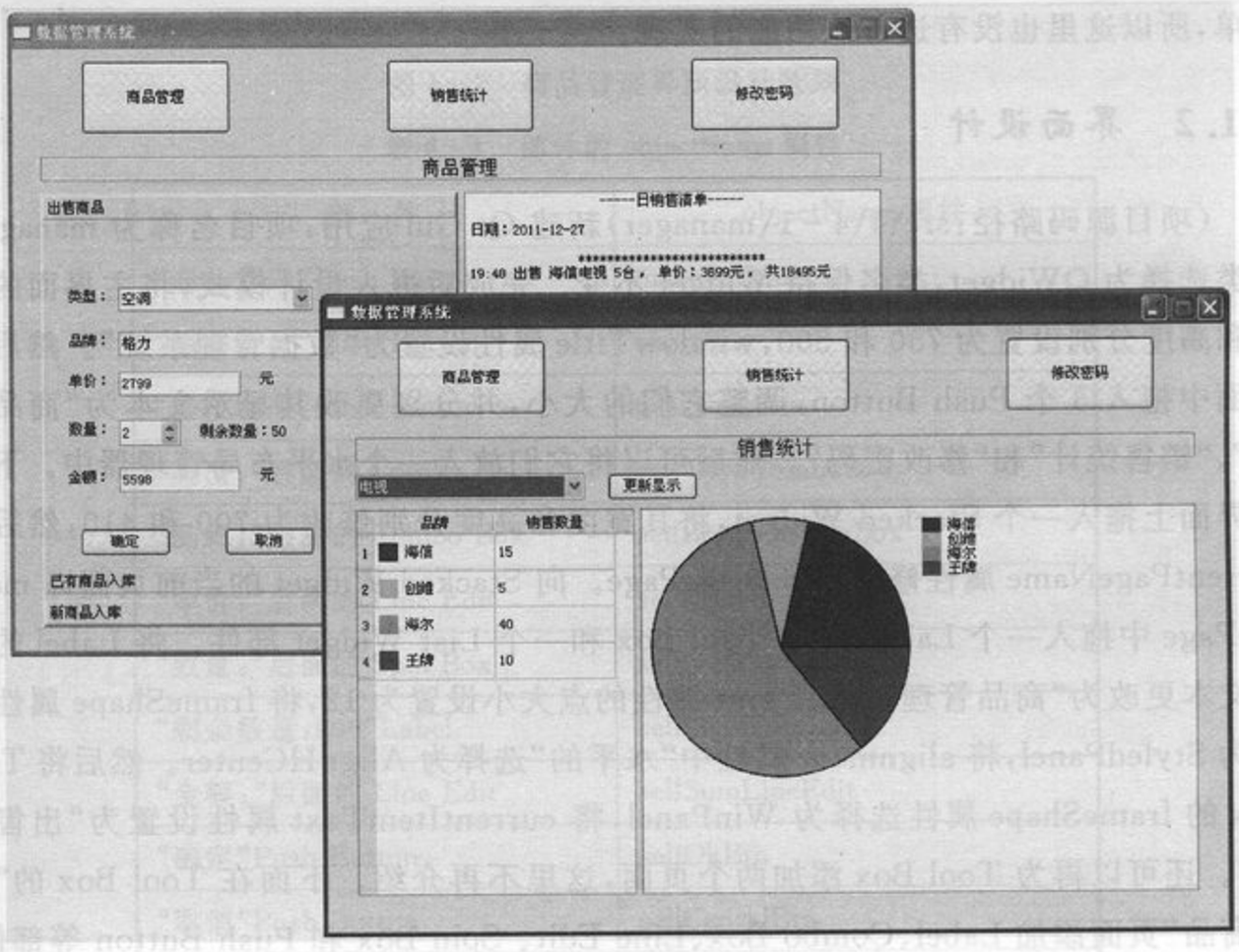



图 4-1　数据管理系统运行界面

# 4.1 功能介绍与界面设计

## 4.1.1 功能介绍

该数据管理系统实现了家电商城商品的入库与销售的管理。为了降低程序的复杂度，这里只使用了电视和空调两种类型家电的简单几条数据进行演示。运行程序后会出现登录界面，需要输入密码才可以登录，该功能会在本章的后面进行讲解。进入主界面后会有3个选项：商品管理、销售统计和修改密码。在商品管理界面实现了商品的销售和入库操作，还显示了当日的销售记录。其中，商品数据信息使用了数据库进行存储，而销售记录使用了XML文档来存储。在销售统计界面通过自定义视图实现了商品销量的图表显示。图4-1中分别显示了商品管理界面和销售统计界面的运行效果。修改密码功能就是对数据库密码表中现有的密码进行修改。

在商品管理界面我们只实现了出售商品和显示日销售清单的功能，因为商品入库的操作和出售商品的操作很相似，读者可以自己实现。而修改密码界面的功能很简单，所以这里也没有进行该功能的实现。

## 4.1.2 界面设计

（项目源码路径：src\4\4-1\manager）新建Qt Gui应用，项目名称为manager，基类选择为QWidget，类名保持Widget不变。完成后进入设计模式，将主界面的宽度和高度分别设置为750和500，windowTitle属性设置为“数据管理系统”。然后向界面中拖入3个Push Button，调整它们的大小，并分别更改其显示文本为“商品管理”、“销售统计”和“修改密码”。然后可以将它们放入一个水平布局管理器中。下面向界面上拖入一个Stacked Widget，将其宽度和高度分别修改为700和410，然后将currentPageName属性修改为managePage。向Stacked Widget的当前页面即managePage中拖入一个Label、一个Tool Box和一个List Widget部件。将Label的显示文本更改为“商品管理”，将其font属性的点大小设置为12，将frameShape属性选择为StyledPanel，将alignment属性中“水平的”选择为AlignHCenter。然后将Tool Box的frameShape属性选择为WinPanel，将currentItemText属性设置为“出售商品”。还可以再为Tool Box添加两个页面，这里不再介绍。下面在Tool Box的“出售商品”页面添加Label、Combo Box、Line Edit、Spin Box和Push Button等部件，最终的效果如图4-2所示。

下面来更改部分部件的objectName属性，如表4-1所列。

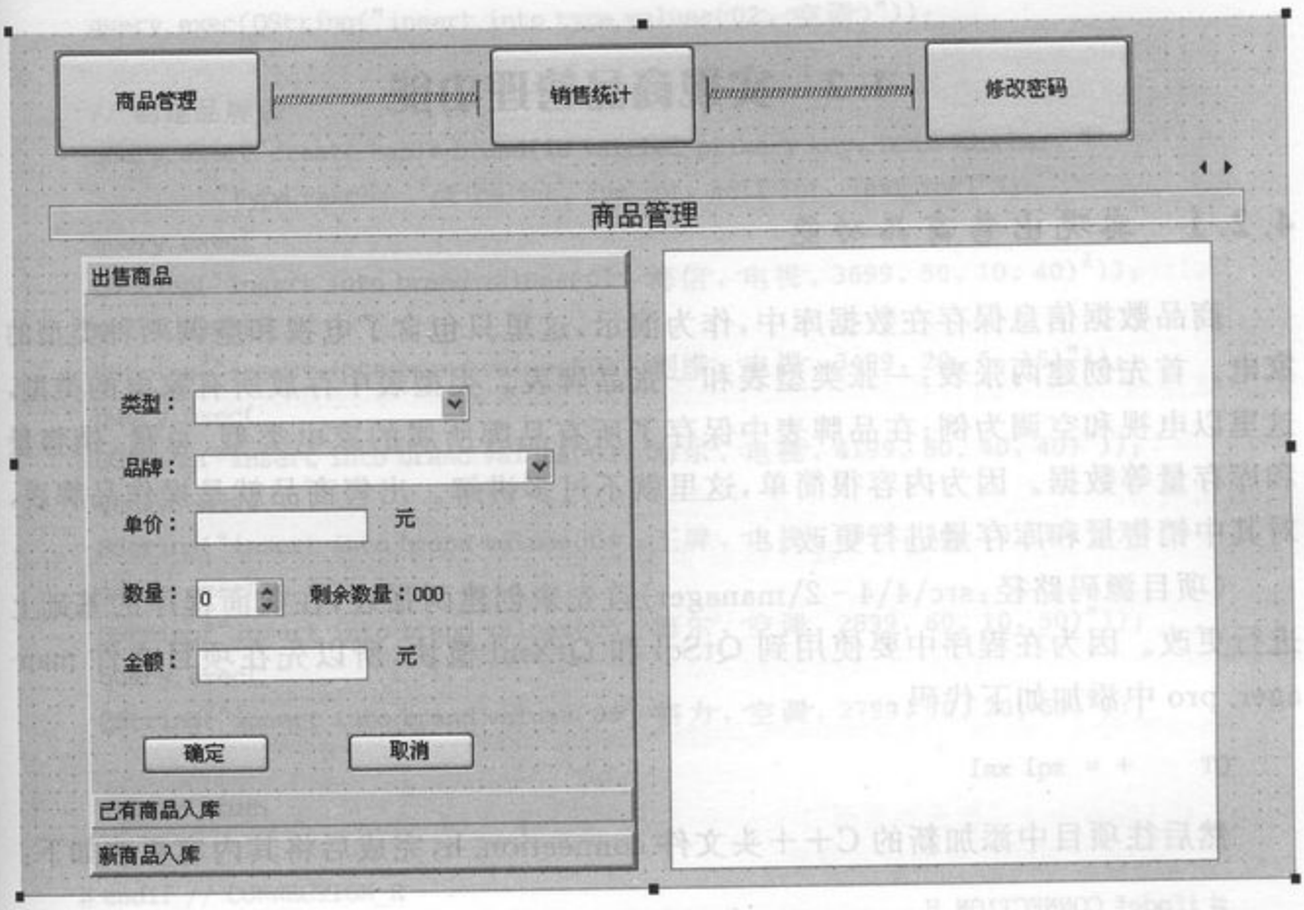



图 4-2　商品管理界面设计效果

**表 4-1　部件的 objectName 属性**

| 部　件 | objectName 属性 |
|---|---|
| “商品管理”Push Button | manageBtn |
| “销售统计”Push Button | chartBtn |
| “修改密码”Push Button | passwordBtn |
| “类型：”后面的 Combo Box | sellTypeComboBox |
| “品牌：”后面的 Combo Box | sellBrandComboBox |
| “单价：”后面的 Line Edit | sellPriceLineEdit |
| “数量：”后面的 Spin Box | sellNumSpinBox |
| “剩余数量：000”Label | sellLastNumLabel |
| “金额：”后面的 Line Edit | sellSumLineEdit |
| “确定”Push Button | sellOkBtn |
| “取消”Push Button | sellCancelBtn |
| 显示日销售清单的 List Widget | dailyList |

## 4.2 实现商品管理功能

### 4.2.1 实现出售商品功能

商品数据信息保存在数据库中，作为演示，这里只包含了电视和空调两种类型的家电。首先创建两张表：一张类型表和一张品牌表。类型表中存放所有家电的类型，这里以电视和空调为例；在品牌表中保存了所有品牌所属的家电类型、总量、销售量和库存量等数据。因为内容很简单，这里就不过多讲解。出售商品就是操作品牌表，对其中销售量和库存量进行更改。

（项目源码路径：src\4\4-2\manager）首先来创建两张表，在前面程序的基础上进行更改。因为在程序中要使用到 QtSql 和 QtXml 模块，所以先在项目文件 manager.pro 中添加如下代码：

```
QT      += sql xml
```

然后往项目中添加新的 C++头文件 connection.h，完成后将其内容更改如下：

```
#ifndef CONNECTION_H
#define CONNECTION_H

#include <QtSql>
#include <QDebug>

static bool createConnection()
{
 QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");

 db.setHostName("yafeilinux");
 db.setDatabaseName("data.db");
 db.setUserName("yafei");
 db.setPassword("123456");
 if (! db.open()) {
  //提示出错
  return false;
 }

 QSqlQuery query;

 // 创建分类表
 query.exec("create table type(id varchar primary key, name varchar)");
 query.exec(QString("insert into type values(0, '请选择类型')"));
 query.exec(QString("insert into type values(01, '电视')"));
```

```
    query.exec(QString("insert into type values(02, 空调)"));

    // 创建品牌表
    query.exec("create table brand(id varchar primary key, name varchar, "
               "type varchar, price int, sum int, sell int, last int)");
    query.exec(
    QString("insert into brand values(01, 海信, 电视, 3699, 50, 10, 40)"));
    query.exec(
    QString("insert into brand values(02, 创维, 电视, 3499, 20, 5, 15)"));
    query.exec(
    QString("insert into brand values(03, 海尔, 电视, 4199, 80, 40, 40)"));
    query.exec(
    QString("insert into brand values(04, 王牌, 电视, 3999, 40, 10, 30)"));
    query.exec(
    QString("insert into brand values(05, 海尔, 空调, 2899, 60, 10, 50)"));
    query.exec(
    QString("insert into brand values(06, 格力, 空调, 2799, 70, 20, 50)"));

    return true;
}
#endif // CONNECTION_H
```

这里创建了两张表 type 和 brand，并且添加了一些初始信息来帮助我们进行功能演示。下面进入 main.cpp 文件，先添加头文件包含：

```
#include <QTextCodec>
#include "connection.h"
```

然后在主函数中第一行代码的下面添加如下代码：

```
QTextCodec::setCodecForTr(QTextCodec::codecForLocale());
QTextCodec::setCodecForCStrings(QTextCodec::codecForLocale());
if(! createConnection()) return 0;
```

这里的 QTextCodec 是用来设置中文字符编码的，因为创建数据库表时里面包含的中文字符使用 QString()进行了编码，所以这里要在调用 createConnection()函数前使用 QTextCodec 来设置编码。

下面进入 widget.cpp 文件，先添加相关的头文件包含。然后在构造函数中添加如下代码：

```
setFixedSize(750, 500);
ui->stackedWidget->setCurrentIndex(0);

QSqlQueryModel *typeModel = new QSqlQueryModel(this);
typeModel->setQuery("select name from type");
ui->sellTypeComboBox->setModel(typeModel);
```

```
QSplitter * splitter = new QSplitter(ui->managePage);
splitter->resize(700, 360);
splitter->move(0, 50);
splitter->addWidget(ui->toolBox);
splitter->addWidget(ui->dailyList);
splitter->setStretchFactor(0, 1);
splitter->setStretchFactor(1, 1);
```

这里先固定了窗口的大小，然后使用 type 表的 name 字段来为类型组合框提供条目，最后为 toolBox 和 dailyList 两个部件进行了布局。现在可以运行程序，会发现类型组合框中已经可以显示所有的家电类型了，如图 4-3 所示。

在类型中选择一种家电，就应该在下面的品牌组合框中显示相应类型的所有品牌。为了实现这个功能，我们从设计模式进入 sellTypeComboBox 部件的 currentIndexChanged(QString)信号对应的槽，然后添加如下代码：

```
void Widget::on_sellTypeComboBox_currentIndexChanged(QString type)
{
  if (type == "请选择类型") {
    // 进行其他部件的状态设置
  } else {
    ui->sellBrandComboBox->setEnabled(true);
    QSqlQueryModel * model = new QSqlQueryModel(this);
    model->setQuery(QString("select name from brand where type='%1'").arg(type));
    ui->sellBrandComboBox->setModel(model);
    ui->sellCancelBtn->setEnabled(true);
  }
}
```

这里先判断选择的是否是“请选择类型”，如果是这个条目，那么就清空其他部件的内容，并将它们设置为不可用，这里的代码我们后面再添加。如果选择了其他条目，就使用选择的家电类型在 brand 表中查找相应的品牌，并在品牌组合框中显示所有的结果。现在运行程序，然后可以选择“电视”类型，这时品牌组合框的内容如图 4-4所示。

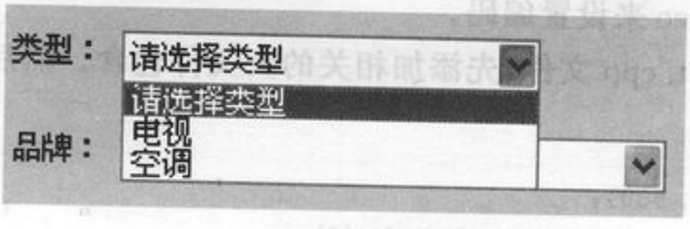



图 4-3　类型组合框的可选条目

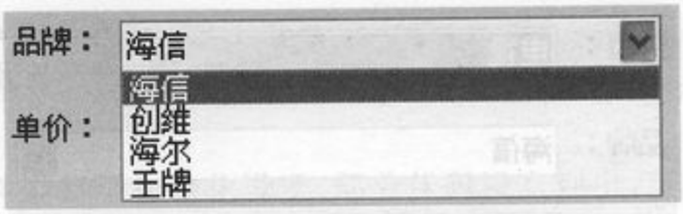



图 4-4　品牌组合框的可选条目

当选择了品牌后需要自动在下面的单价和剩余数量处显示相应的信息。从设计模式进入 sellBrandComboBox 部件的 currentIndexChanged(QString)信号对应的槽,然后添加如下代码:

```
void Widget::on_sellBrandComboBox_currentIndexChanged(QString brand)
{
    ui->sellNumSpinBox->setValue(0);
    ui->sellNumSpinBox->setEnabled(false);
    ui->sellSumLineEdit->clear();
    ui->sellSumLineEdit->setEnabled(false);
    ui->sellOkBtn->setEnabled(false);

    QSqlQuery query;
    query.exec(QString("select price from brand where name='%1' and type='%2'").arg"
"(brand).arg(ui->sellTypeComboBox->currentText()));
    query.next();
    ui->sellPriceLineEdit->setEnabled(true);
    ui->sellPriceLineEdit->setReadOnly(true);
    ui->sellPriceLineEdit->setText(query.value(0).toString());

    query.exec(QString("select last from brand where name='%1' and type='%2'")"
        ".arg(brand).arg(ui->sellTypeComboBox->currentText()));
    query.next();
    int num = query.value(0).toInt();
    if (num == 0) {
      QMessageBox::information(this, tr("提示"), tr("该商品已经售完!"),QMessageBox::Ok);
    } else {
      ui->sellNumSpinBox->setEnabled(true);
      ui->sellNumSpinBox->setMaximum(num);
      ui->sellLastNumLabel->setText(tr("剩余数量:%1").arg(num));
      ui->sellLastNumLabel->setVisible(true);
    }
}
```

这里先设置了几个部件的状态,然后查询了当前类型的品牌单价和剩余数量并进行了显示。注意,这里将购买数量选择框的上限设置为了当前的剩余数量。下面运行程序,效果如图 4-5 所示。

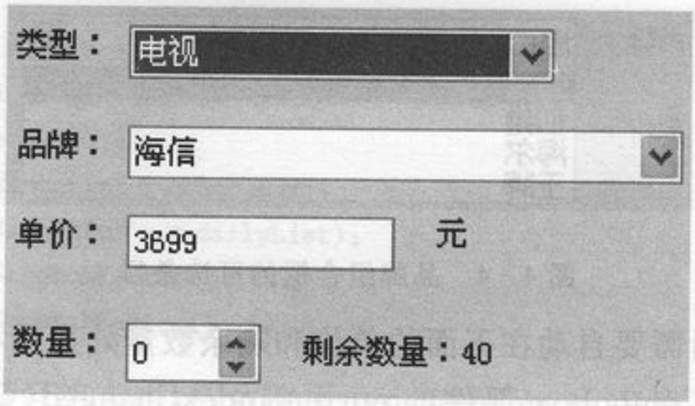



图 4-5　显示单价和剩余数量信息

更改了购买数量后会自动显示总的金额。从设计模式进入 sellNumSpinBox 部件的 valueChanged(int)信号对应的槽，然后添加如下代码：

```
void Widget::on_sellNumSpinBox_valueChanged(int value)
{
  if (value == 0) {
    ui->sellSumLineEdit->clear();
    ui->sellSumLineEdit->setEnabled(false);
    ui->sellOkBtn->setEnabled(false);
  } else {
    ui->sellSumLineEdit->setEnabled(true);
    ui->sellSumLineEdit->setReadOnly(true);
    qreal sum = value * ui->sellPriceLineEdit->text().toInt();
    ui->sellSumLineEdit->setText(QString::number(sum));
    ui->sellOkBtn->setEnabled(true);
  }
}
```

这里获取了单价和购买数量并计算出总价，然后进行了显示，程序运行效果如图 4-6 所示。下面从设计模式进入“取消”按钮的单击信号对应的槽，更改如下：

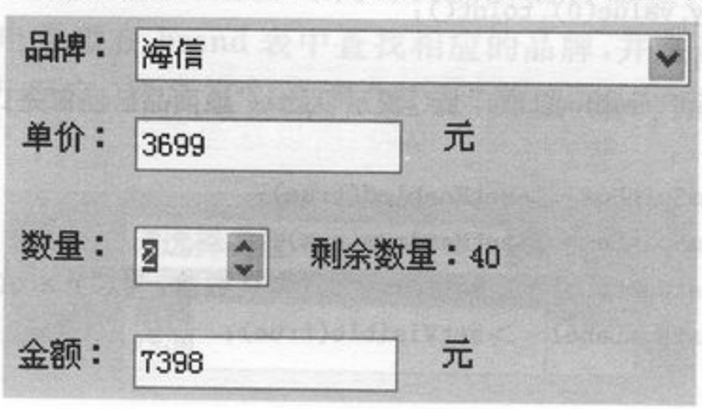



图 4-6　显示购买商品的总价

```
void Widget::on_sellCancelBtn_clicked()
{
  ui->sellTypeComboBox->setCurrentIndex(0);
```

```
    ui->sellBrandComboBox->clear();
    // 省略了部分代码
}
```

这里对各个部件进行了初始化设置，部分代码没有列出，可以下载本书的源码进行参考。下面返回到前面的 on_sellTypeComboBox_currentIndexChanged()槽中，在“进行其他部件的状态设置”注释的地方添加该槽的调用：

```
on_sellCancelBtn_clicked();
```

然后在构造函数末尾也添加该槽的调用，用来初始化界面。最后进入“确定”按钮的单击信号对应的槽，更改如下：

```
void Widget::on_sellOkBtn_clicked()
{
    QString type = ui->sellTypeComboBox->currentText();
    QString name = ui->sellBrandComboBox->currentText();
    int value = ui->sellNumSpinBox->value();
    // cellNumSpinBox 的最大值就是以前的剩余量
    int last = ui->sellNumSpinBox->maximum() - value;

    QSqlQuery query;
    // 获取以前的销售量
    query.exec(QString("select sell from brand where name='%1' and type='%2'").arg"
"(name).arg(type));
    query.next();
    int sell = query.value(0).toInt() + value;

    // 事务操作
    QSqlDatabase::database().transaction();
    bool rtn = query.exec(
      QString("update brand set sell = %1,last = %2 where name='%3' and type='%4'")"
".arg(sell).arg(last).arg(name).arg(type));

    if (rtn) {
      QSqlDatabase::database().commit();
      QMessageBox::information(this, tr("提示"), tr("购买成功!"),QMessageBox::Ok);
      // writeXml();
      // showDailyList();
      on_sellCancelBtn_clicked();
    } else {
      QSqlDatabase::database().rollback();
    }
}
```

这里获取了以前的销售量和剩余量，然后计算出当前的销售量和剩余量并修改

数据库中的数据。注意在购买成功后还调用了writeXml()和showDailyList()函数来显示日销售清单，这两个函数会在后面的内容中讲到，为了让程序现在可以正常运行，先注释掉这两个函数。运行程序，然后进行一些销售操作。

程序中还应有商品入库的操作，也是对数据库进行更改。该功能的实现与前面讲的商品销售操作的实现很相似，这里就不再讲解。

### 4.2.2 显示日销售清单

商品管理界面右侧的List Widget部件用来显示每天的商品销售记录。商品的销售记录使用XML文档来保存，而在List Widget中只显示当日的销售情况。在该XML文档中按日期和时间存储了所出售的商品的数量和金额等信息，下面是该文档的一个例子：

```
<? xml version = 1.0 encoding = UTF - 8? >
<日销售清单>
  <日期 date = "2011 - 12 - 28">
    <时间 time = "10:52">
      <类型>空调</类型>
      <品牌>海尔</品牌>
      <单价>2899</单价>
      <数量>1</数量>
      <金额>2899</金额>
    </时间>
    <时间 time = "10:52">
      <类型>电视</类型>
      <品牌>创维</品牌>
      <单价>3499</单价>
      <数量>1</数量>
      <金额>3499</金额>
    </时间>
  </日期>
  <日期 date = "2011 - 12 - 29">
    <时间 time = "15:21">
      <类型>电视</类型>
      <品牌>王牌</品牌>
      <单价>3999</单价>
      <数量>3</数量>
      <金额>11997</金额>
    </时间>
  </日期>
</日销售清单>
```

在前面的程序中已经看到了，每成功出售一次商品就将相关信息写入到XML

文件，然后更新显示。在进行相关操作之前，我们需要先创建 XML 文件。

（项目源码路径：src\4\4 - 3\manager）首先在 connection.h 文件中添加头文件，然后在其中添加一个创建 XML 文件的函数的定义：

```
static bool createXml()
{
  QFile file("data.xml");
  if(file.exists()) return true;
  if(! file.open(QIODevice::WriteOnly | QIODevice::Truncate))
  return false;
  QDomDocument doc;
  QDomProcessingInstruction instruction;
  instruction = doc.createProcessingInstruction(
              "xml","version = \"1.0\" encoding = \"UTF - 8\"");
  doc.appendChild(instruction);
  QDomElement root = doc.createElement(QString("日销售清单"));
  doc.appendChild(root);
  QTextStream out(&file);
  doc.save(out,4);
  file.close();
  return true;
}
```

这里创建了一个 data.xml 文件，并在其中添加了根元素。为了不重复创建该文件，在创建前先使用了 QFile::exists()函数来判断该文件是否已经被创建。下面在主函数中调用该函数，到 main.cpp 文件中将以前的 createConnection()函数调用的代码更改为：

```
if(! createConnection() || ! createXml()) return 0;
```

然后到 widget.h 文件中，先添加头文件包含，再定义一个 public 函数和枚举变量用来获取当前的时间和日期：

```
enum DateTimeType{Time, Date, DateTime};
QString getDateTime(DateTimeType type);
```

下面添加一个私有对象定义：

```
QDomDocument doc;
```

再添加几个私有函数声明：

```
private:
  bool docRead();
  bool docWrite();
  void writeXml();
  void createNodes(QDomElement &date);
```

```
void showDailyList();
```

下面到 widget.cpp 文件中添加这几个函数的定义。首先来添加获取时间和日期的函数的定义：

```
QString Widget::getDateTime(Widget::DateTimeType type)
{
  QDateTime datetime = QDateTime::currentDateTime();
  QString date = datetime.toString("yyyy-MM-dd");
  QString time = datetime.toString("hh:mm");
  QString dateAndTime = datetime.toString("yyyy-MM-dd dddd hh:mm");
  if(type == Date) return date;
  else if(type == Time) return time;
  else return dateAndTime;
}
```

这个函数使用了我们自定义的 DateTimeType 枚举变量为参数来返回需要使用的格式化的时间或日期，创建该函数的目的就是为了方便其他函数调用。下面来添加读取和写入 XML 文件的函数，这两个函数也是为了可以在其他地方重复使用：

```
// 读取 XML 文档
bool Widget::docRead()
{
  QFile file("data.xml");
  If (! file.open(QIODevice::ReadOnly))
    return false;
  if (! doc.setContent(&file)) {
    file.close();
    return false;
  }
  file.close();
  return true;
}
// 写入 xml 文档
bool Widget::docWrite()
{
  QFile file("data.xml");
  If (! file.open(QIODevice::WriteOnly | QIODevice::Truncate))
    return false;
  QTextStream out(&file);
  doc.save(out,4);
  file.close();
  return true;
}
```

下面添加修改 XML 文档的 writeXml()函数，该函数在销售商品成功时被调用，

将销售商品信息保存到 XML 文件中：

```
void Widget::writeXml()
{
  // 先从文件读取
  if (docRead()) {
    QString currentDate = getDateTime(Date);
    QDomElement root = doc.documentElement();
    // 根据是否有日期节点进行处理
    if (! root.hasChildNodes()) {
      QDomElement date = doc.createElement(QString("日期"));
      QDomAttr curDate = doc.createAttribute("date");
      curDate.setValue(currentDate);
      date.setAttributeNode(curDate);
      root.appendChild(date);
      createNodes(date);
    } else {
      QDomElement date = root.lastChild().toElement();
      // 根据是否已经有今天的日期节点进行处理
      if (date.attribute("date") = = currentDate) {
        createNodes(date);
      } else {
        QDomElement date = doc.createElement(QString("日期"));
        QDomAttr curDate = doc.createAttribute("date");
        curDate.setValue(currentDate);
        date.setAttributeNode(curDate);
        root.appendChild(date);
        createNodes(date);
      }
    }
    // 写入到文件
    docWrite();
  }
}
```

这里先判断了是否有日期节点，如果是第一次操作，那么没有日期节点，需要重新创建。如果有日期节点，就判断是否有今天的日期节点，如果没有，就创建今天的日期节点，否则在今天的日期节点下直接添加销售商品的信息。这里使用了 createNodes()函数来创建销售商品信息的节点，下面添加该函数的定义：

```
void Widget::createNodes(QDomElement &date)
{
  QDomElement time = doc.createElement(QString("时间"));
  QDomAttr curTime = doc.createAttribute("time");
  curTime.setValue(getDateTime(Time));
```

```
    time.setAttributeNode(curTime);
    date.appendChild(time);
    QDomElement type = doc.createElement(QString("类型"));
    QDomElement brand = doc.createElement(QString("品牌"));
    QDomElement price = doc.createElement(QString("单价"));
    QDomElement num = doc.createElement(QString("数量"));
    QDomElement sum = doc.createElement(QString("金额"));
    QDomText text;
    text = doc.createTextNode(QString("%1")
                    .arg(ui->sellTypeComboBox->currentText()));
    type.appendChild(text);
    text = doc.createTextNode(QString("%1")
                    .arg(ui->sellBrandComboBox->currentText()));
    brand.appendChild(text);
    text = doc.createTextNode(QString("%1")
                    .arg(ui->sellPriceLineEdit->text()));
    price.appendChild(text);
    text = doc.createTextNode(QString("%1")
                    .arg(ui->sellNumSpinBox->value()));
    num.appendChild(text);
    text = doc.createTextNode(QString("%1")
                    .arg(ui->sellSumLineEdit->text()));
    sum.appendChild(text);
    time.appendChild(type);
    time.appendChild(brand);
    time.appendChild(price);
    time.appendChild(num);
    time.appendChild(sum);
}
```

最后添加显示日销售清单的函数的定义，在该函数中读取了XML文档中今天的销售记录信息并以一定的格式进行显示：

```
void Widget::showDailyList()
{
  ui->dailyList->clear();
  if (docRead()) {
    QDomElement root = doc.documentElement();
    QString title = root.tagName();
    QListWidgetItem *titleItem = new QListWidgetItem;
    titleItem->setText(QString("-----%1------").arg(title));
    titleItem->setTextAlignment(Qt::AlignCenter);
    ui->dailyList->addItem(titleItem);
    if (root.hasChildNodes()) {
      QString currentDate = getDateTime(Date);
```

```
            QDomElement dateElement = root.lastChild().toElement();
            QString date = dateElement.attribute("date");
            if (date == currentDate) {
                ui->dailyList->addItem("");
                ui->dailyList->addItem(QString("日期：%1").arg(date));
                ui->dailyList->addItem("");
                QDomNodeList children = dateElement.childNodes();
                // 遍历当日销售的所有商品
                for (int i = 0; i<children.count(); i++) {
                    QDomNode node = children.at(i);
                    QString time = node.toElement().attribute("time");
                    QDomNodeList list = node.childNodes();
                    QString type = list.at(0).toElement().text();
                    QString brand = list.at(1).toElement().text();
                    QString price = list.at(2).toElement().text();
                    QString num = list.at(3).toElement().text();
                    QString sum = list.at(4).toElement().text();
                    QString str = time + " 出售 " + brand + type
                            + " " + num + "台，" + "单价：" + price
                            + "元，共" + sum + "元";
                    QListWidgetItem *tempItem = new QListWidgetItem;
                    tempItem->setText("**************************");
                    tempItem->setTextAlignment(Qt::AlignCenter);
                    ui->dailyList->addItem(tempItem);
                    ui->dailyList->addItem(str);
                }
            }
        }
    }
}
```

下面将 on_sellOkBtn_clicked()函数中的 writeXml()和 showDailyList()函数调用的注释去掉。然后再到构造函数中添加 showDailyList()函数的调用，即添加如下代码：

```
showDailyList();
```

下面可以运行程序，然后进行商品的销售操作，完成后会在界面右边的列表部件中显示出销售记录，效果如图 4-1 所示。

## 4.3　显示销售统计图表

在销售统计界面分别使用了数据表格和饼状图来显示商品的销售数量信息，如图 4-1 所示。其中饼状图是通过自定义视图来实现的，也可以参考 Qt 自带的

Chart Example示例程序。

(项目源码路径:src\4\4-4\manager)向项目中添加新的C++类,类名设置为PieView,基类设置为QAbstractItemView,类型信息选择“继承自QWidget”。完成后在pieview.h和pieview.cpp文件中添加代码,因为篇幅有限,这里就不再列出所有代码。与视图显示有关的最主要的是其中的paintEvent()函数,下面来看一下该函数的定义:

```
void PieView::paintEvent(QPaintEvent * event)
{
  QItemSelectionModel * selections = selectionModel();
  QStyleOptionViewItem option = viewOptions();

  QBrush background = option.palette.base();
  QPen foreground(option.palette.color(QPalette::WindowText));

  QPainter painter(viewport());
  painter.setRenderHint(QPainter::Antialiasing);

  painter.fillRect(event->rect(), background);
  painter.setPen(foreground);

  QRect pieRect = QRect(margin, margin, pieSize, pieSize);

  if (validItems > 0) {
    // 绘制圆形饼状图
    painter.save();
    painter.translate(pieRect.x() - horizontalScrollBar()->value(),pieRect.y() -
verticalScrollBar()->value());
    painter.drawEllipse(0, 0, pieSize, pieSize);
    double startAngle = 0.0;
    int row;
    for (row = 0; row < model()->rowCount(rootIndex()); ++row) {
      QModelIndex index = model()->index(row, 1, rootIndex());
      double value = model()->data(index).toDouble();
      if (value > 0.0) {
        double angle = 360 * value/totalValue;
        QModelIndex colorIndex = model()->index(row, 0, rootIndex());
        QColor color = QColor(model()->data(colorIndex,
                              Qt::DecorationRole).toString());
        if (currentIndex() == index)
          painter.setBrush(QBrush(color, Qt::Dense4Pattern));
        else if (selections->isSelected(index))
          painter.setBrush(QBrush(color, Qt::Dense3Pattern));
        else
```

```
            painter.setBrush(QBrush(color));
            painter.drawPie(0, 0, pieSize, pieSize, int(startAngle * 16),
                            int(angle * 16));
            startAngle += angle;
        }
    }
    painter.restore();

    // 绘制饼状图旁边的图示
    int keyNumber = 0;
    for (row = 0; row < model()->rowCount(rootIndex()); ++row) {
        QModelIndex index = model()->index(row, 1, rootIndex());
        double value = model()->data(index).toDouble();
        if (value > 0.0) {
            QModelIndex labelIndex = model()->index(row, 0, rootIndex());
            QStyleOptionViewItem option = viewOptions();
            option.rect = visualRect(labelIndex);
            if (selections->isSelected(labelIndex))
                option.state |= QStyle::State_Selected;
            if (currentIndex() == labelIndex)
                option.state |= QStyle::State_HasFocus;
            itemDelegate()->paint(&painter, option, labelIndex);
            keyNumber++;
        }
    }
}
```

先要说明一下，我们给这个视图传入的数据模型需要有两个字段，第一个是品牌名称，它还有一个颜色数据角色；第二个是销售数量。这个数据模型会在后面的代码中进行创建，图 4-7 是该模型的一个实例。这里的代码中从模型的第二个字段中获取了销售数量值：

```
QModelIndex index = model()->index(row, 1, rootIndex());
double value = model()->data(index).toDouble();
```

然后根据该值的大小来确定扇形的大小。而颜色是从模型的第一个字段中获取的：

```
QModelIndex colorIndex = model()->index(row, 0, rootIndex());
QColor color = QColor(model()->data(colorIndex,
                      Qt::DecorationRole).toString());
```

使用获取的颜色来填充相应的扇形。这样视图最终显示为多个不同颜色扇形组成的圆形。

下面来设计销售统计的界面。先进入设计模式，单击 Stacked Widget 右上角的小箭头进入下一页。然后更改当前页的名称，即将 Stacked Widget 的 currentPageName 属性更改为 chartPage。完成后向当前页上拖入一个 Push Button、一个 Combo Box 和一个 Label，最终效果如图 4-8 所示。这里需要将 ComboBox 和 Push Button 的 objectName 分别更改为 typeComboBox 和 updateBtn。

| | 品牌 | 销售数量 |
|---|---|---|
| 1 | 海信 | 10 |
| 2 | 创维 | 5 |
| 3 | 海尔 | 40 |
| 4 | 王牌 | 10 |

图 4-7　数据模型实例


图 4-8　销售统计界面设计效果

下面进入 widget.h 文件，先添加类的前置声明：

```
class QStandardItemModel;
```

然后添加一个私有对象定义：

```
QStandardItemModel * chartModel;
```

再添加两个私有函数声明：

```
void createChartModelView();
void showChart();
```

现在进入 widget.cpp 文件中，先添加 #include "pieview.h"头文件包含，然后在构造函数中添加如下代码：

```
ui->typeComboBox->setModel(typeModel);
createChartModelView();
```

这里的 createChartModelView()函数是用来创建销售数据的模型和视图的，下面添加该函数的定义：

```
void Widget::createChartModelView()
{
 chartModel = new QStandardItemModel(this);
 chartModel->setColumnCount(2);
 chartModel->setHeaderData(0, Qt::Horizontal, QString("品牌"));
 chartModel->setHeaderData(1, Qt::Horizontal, QString("销售数量"));

 QSplitter * splitter = new QSplitter(ui->chartPage);
 splitter->resize(700, 320);
 splitter->move(0, 80);
 QTableView * table = new QTableView;
```

```
    PieView * pieChart = new PieView;
    splitter->addWidget(table);
    splitter->addWidget(pieChart);
    splitter->setStretchFactor(0, 1);
    splitter->setStretchFactor(1, 2);

    table->setModel(chartModel);
    pieChart->setModel(chartModel);

    QItemSelectionModel * selectionModel = new QItemSelectionModel(chartModel);
    table->setSelectionModel(selectionModel);
    pieChart->setSelectionModel(selectionModel);
}
```

这里先创建了一个包含两个列的数据模型，然后创建了一个表格视图和一个饼状图视图，并对它们进行了布局。然后为视图指定了模型，并设置了两个视图共享选择模型。现在来为数据模型添加数据，这是在 showChart()函数中完成的，下面添加它的定义：

```
void Widget::showChart()
{
    QSqlQuery query;
    query.exec(QString("select name,sell from brand where type='%1'")
            .arg(ui->typeComboBox->currentText()));

    chartModel->removeRows(0, chartModel->rowCount(QModelIndex()),
                           QModelIndex());
    int row = 0;

    while(query.next()) {
        int r = qrand() % 256;
        int g = qrand() % 256;
        int b = qrand() % 256;

        chartModel->insertRows(row, 1, QModelIndex());

        chartModel->setData(chartModel->index(row, 0, QModelIndex()),
                            query.value(0).toString());
        chartModel->setData(chartModel->index(row, 1, QModelIndex()),
                            query.value(1).toInt());
        chartModel->setData(chartModel->index(row, 0, QModelIndex()),
                            QColor(r, g, b), Qt::DecorationRole);
        row++;
    }
}
```

这里从数据库brand表中查出了对应类型的品牌及其销售数量信息，然后将它们添加到了创建的数据模型中，这里还为每一个品牌产生了一个随机的颜色。

下面分别从设计模式进入typeComboBox的currentIndexChanged(QString)信号对应的槽和updateBtn的clicked()信号对应的槽，将它们更改如下：

```
void Widget::on_typeComboBox_currentIndexChanged(QString type)
{
  if (type != "请选择类型")
    showChart();
}

void Widget::on_updateBtn_clicked()
{
  if (ui->typeComboBox->currentText() != "请选择类型")
    showChart();
}
```

为了可以使用“商品管理”按钮和“销售统计”按钮来切换界面，还需要分别从设计模式转到这两个按钮的单击信号对应的槽，并更改如下：

```
// 商品管理按钮
void Widget::on_manageBtn_clicked()
{
  ui->stackedWidget->setCurrentIndex(0);
}
// 销售统计按钮
void Widget::on_chartBtn_clicked()
{
  ui->stackedWidget->setCurrentIndex(1);
}
```

现在可以运行程序了，效果如图4-1所示。

## 4.4 添加登录界面

数据管理系统一般都有一个登录界面，需要输入正确的密码才可以登录。在我们的实例中将密码保存在了数据库的密码表中，然后在登录的时候通过读取现有的密码来判断用户输入的密码是否正确。

下面先来创建密码表。(项目源码路径：src\4\4-5\manager)在connection.h文件的createConnection()函数中添加如下代码：

```
query.exec("create table password(pwd varchar primary key)");
query.exec("insert into password values(123456)");
```

这里创建了一个 password 表，然后将初始密码设置为了“123456”。下面来添加登录对话框类。向项目中添加新的 Qt 设计师界面类，模板选择 Dialog without Buttons，类名设置为 LoginDialog。完成后往界面上拖入 Push Button、Line Edit 和 Label 等部件，最终效果如图 4-9 所示。将 Line Edit 的 objectName 属性更改为 pwdLineEdit，然后将 echoMode 选择为 Password；将“登录”按钮的 objectName 属性设置为 loginBtn，“退出”按钮的 objectName 属性设置为 quitBtn。

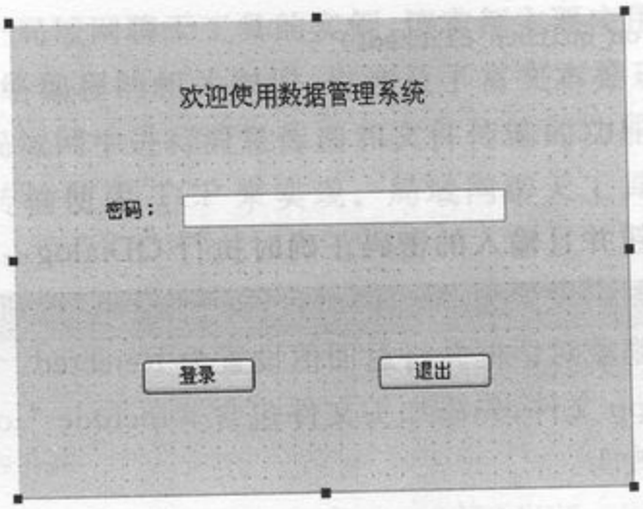



图 4-9　登录界面设计效果

下面进入 logindialog.cpp 文件中，先添加头文件包含，然后在构造函数中添加如下代码：

```
setFixedSize(400, 300);
setWindowTitle(tr("登录"));
ui->pwdLineEdit->setFocus();
ui->loginBtn->setDefault(true);
```

下面从设计模式分别进入“登录”按钮和“退出”按钮的单击信号对应的槽，更改如下：

```
void LoginDialog::on_loginBtn_clicked()
{
  if (ui->pwdLineEdit->text().isEmpty()) {
    QMessageBox::information(this, tr("请输入密码"),
                             tr("请先输入密码再登录!"), QMessageBox::Ok);
    ui->pwdLineEdit->setFocus();
  } else {
    QSqlQuery query;
    query.exec("select pwd from password");
    query.next();
    if (query.value(0).toString() == ui->pwdLineEdit->text()) {
      QDialog::accept();
    } else {
```

```
            QMessageBox::warning(this, tr("密码错误"),
                                 tr("请输入正确的密码再登录!"), QMessageBox::Ok);
            ui->pwdLineEdit->clear();
            ui->pwdLineEdit->setFocus();
        }
    }
}

void LoginDialog::on_quitBtn_clicked()
{
    QDialog::reject();
}
```

当按下“登录”按钮并且输入的密码正确时执行 QDialog::accept()函数，该函数会隐藏对话框并将返回码设置为 Accepted；当按下“退出”按钮时执行 QDialog::reject()函数，该函数会隐藏对话框并将返回值设置为 Rejected。

下面进入 main.cpp 文件，先添加头文件包含 #include "logindialog.h"，然后将主函数的部分代码更改为：

```
Widget w;
LoginDialog dlg;
if (dlg.exec() == QDialog::Accepted) {
    w.show();
    return a.exec();
} else {
    return 0;
}
```

这里使用了对话框的返回值来进行判断，如果返回值为 QDialog::Accepted，则显示主界面，否则退出程序。在实例中还有修改密码的功能，它是通过修改 password 表来实现的，因为这个很容易实现，这里不再讲解。到这里整个实例就完成了，可以运行程序查看一下效果。

## 4.5 小 结

通过本章的学习，读者应该掌握如何在实际应用中使用数据库和 XML 文档。数据库和 XML 文档在存储和显示数据方面各具特长，可以灵活地选择使用。如果要使用特殊的方式来显示数据，比如使用饼状图或者直方图，那么可以使用自定义视图。

该实例中只讲解了最重要的一些功能，其他功能可以自己试着添加，也可以参考我们网站上提供的数据管理系统开源软件。

# 第 5 章　局域网聊天工具

这一章将讲述一个局域网聊天工具的实例，该实例主要应用了《Qt Creator 快速入门》网络通信篇中网络编程的相关知识，也涉及了富文本处理、输入输出等知识。该程序主要实现了在局域网中进行消息传递和文件传输的功能，其中消息传递使用 UDP 来实现，而文件传输使用 TCP 来实现。局域网聊天工具的主界面如图 5－1 所示。

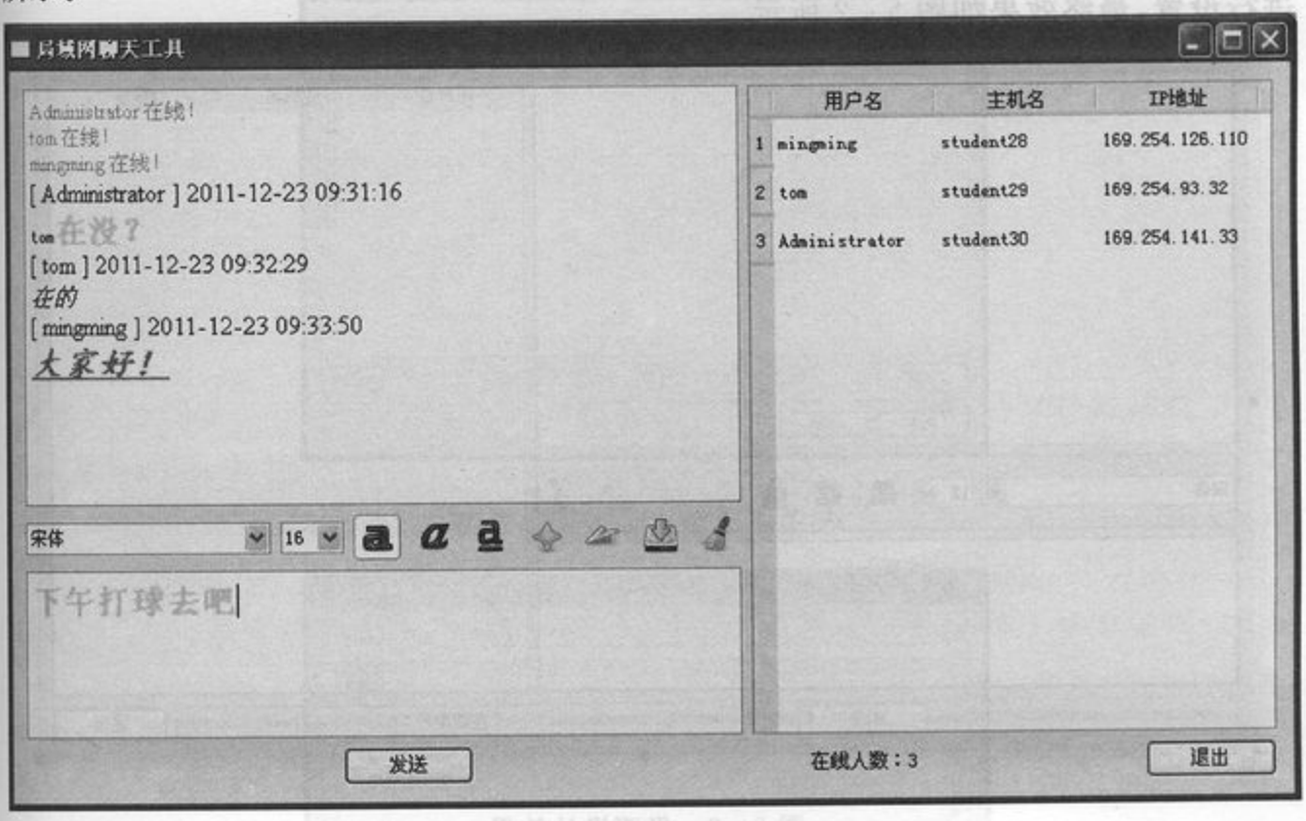



图 5－1　局域网聊天工具主界面

根据要实现的功能，我们分几个步骤来设计该程序。首先是设计界面，实现登录用户信息的显示，接着添加聊天功能，添加传输文件功能，最后进行聊天消息的字体设置等其他优化。整个程序的框架很明了，这里就不再进行过多讲解。每个功能模块的实现原理会在相应的小节详细说明。该实例主要应用了网络编程的相关知识，所以建议先学习《Qt Creator 快速入门》中的第 18 章。也可以参考 Qt 自带的 Network Chat Example 示例程序。

## 5.1 界面设计

首先设计应用程序的界面。(项目源码路径:src\5\5-1\chat)新建Qt Gui应用,项目名称为chat,基类选择QWidget,类名为Widget。完成后在chat.pro文件中添加如下一行代码并按下Ctrl+S保存该文件:

```
QT += network
```

下面向项目添加资源文件并向其中添加一些图标图片用来设计界面使用。然后双击widget.ui文件进入设计模式,先将界面的宽度、高度属性分别设置为800和450,再将其windowTitle属性设置为"局域网聊天工具"。下面向界面上拖入部件并进行设置,最终效果如图5-2所示。

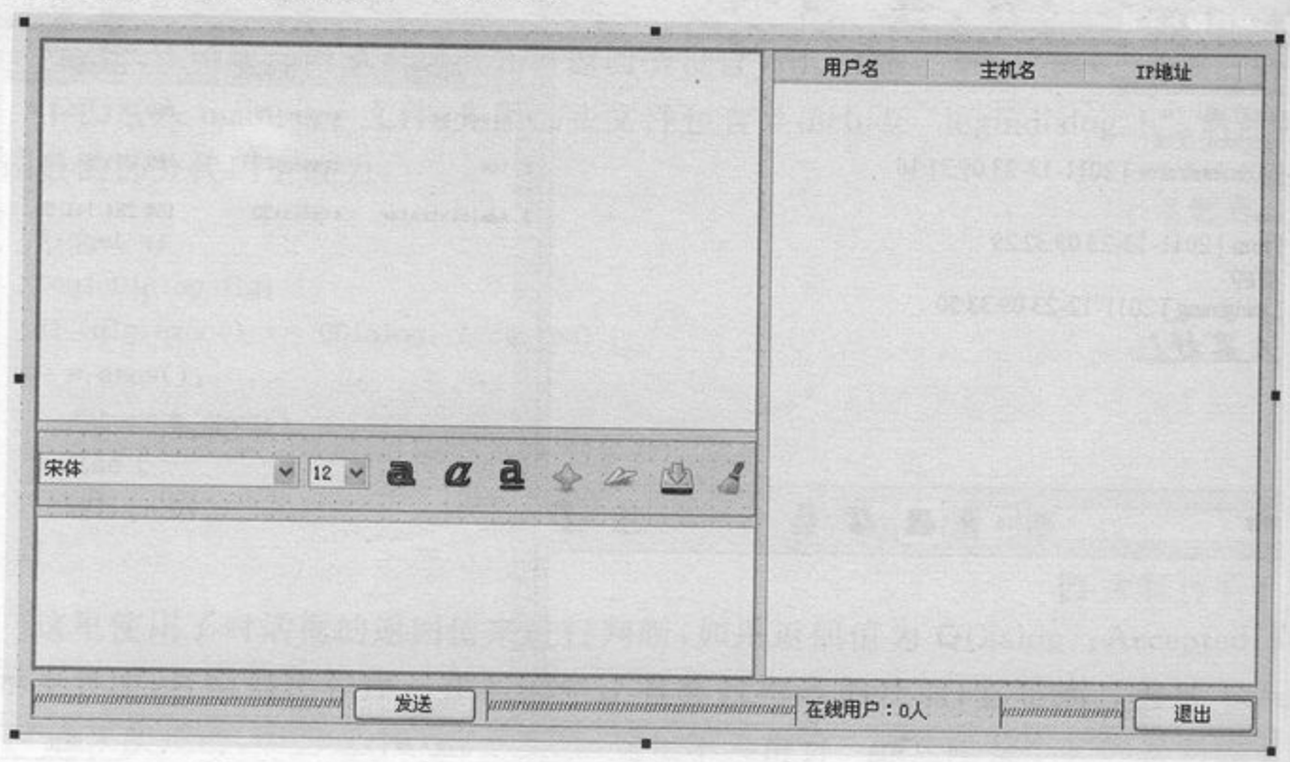



图5-2 界面设计效果

左上角是一个Text Browser部件,主要用来显示用户的聊天记录,将其objectName属性更改为messageBrowser。下面的一行小部件分别是一个Font Combo Box、一个Combo Box和7个Tool Button部件。其中,Font Combo Box部件用来选择字体族,其objectName采用默认的fontComboBox即可,不进行更改;Combo Box用来设置字体的大小,这里设置可选区间为9—22,双击该部件,在弹出的窗口中按"+"号图标新添项目,如图5-3所示。然后将该部件的objectName属性更改为sizeComboBox,currentIndex属性设置为3;而对于后面的7个Tool Button部件,先对它们进行属性更改:宽度和高度均设置为30,iconSize的宽度和高度均设置为22,然后选中autoRaise属性,对前3个Tool Button选中checkable属性。完成后再

分别更改它们的 icon 图标以及 objectName 属性，它们的 objectName 依次更改为：boldToolBtn、italicToolBtn、underlineToolBtn、colorToolBtn、sendToolBtn、saveToolBtn 和 clearToolBtn。再将它们的 toolTip 属性依次更改为“加粗”、“倾斜”、“下划线”、“更改字体颜色”、“传输文件”、“保存聊天记录”和“清空聊天记录”。下面是一个 Text Edit 部件，用来输入要发送的消息，将其 objectName 属性更改为 messageTextEdit。界面右边是一个 Table Widget 部件，用来显示登录的用户列表。先将其 objectName 属性更改为 userTableWidget，再将 selectionMode 属性选择为 SingleSelection，将 selectionBehavior 选择为 SelectRows，然后取消选中的 showGrid。下面双击该部件，在弹出的对话框中添加“用户名”、“主机名”和“IP 地址”3 个列，如图 5-4 所示。

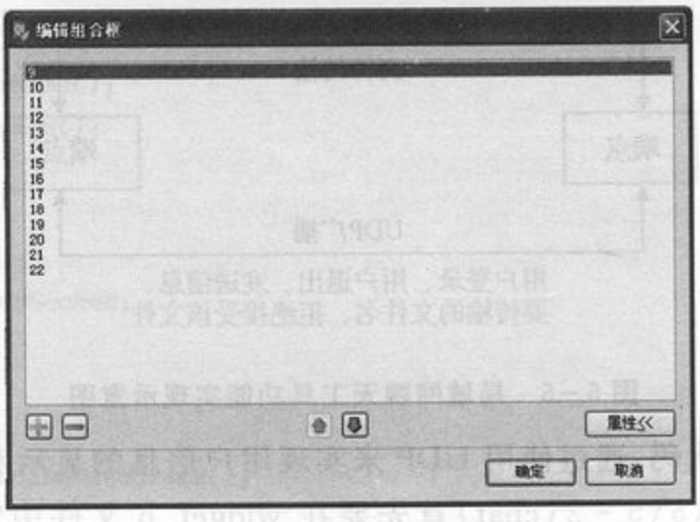



图 5-3　添加字体大小条目

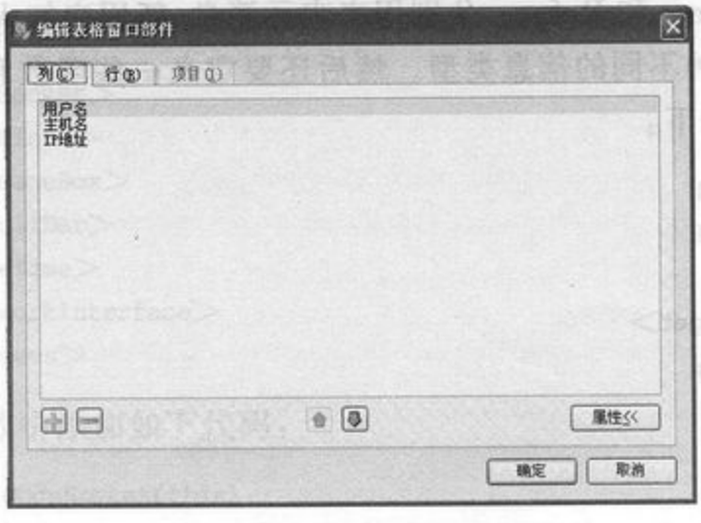



图 5-4　添加表格部件列项目

最后将下面的“发送”按钮、“在线用户”标签和“退出”按钮的 objectName 分别更改为 sendButton、userNumLabel 和 exitButton。也可以将这些部件使用布局管理器进行管理，这样界面会显得更加整齐。

## 5.2 实现聊天功能

在这里先要明确一下，与《Qt Creator快速入门》中第18章的例子不同，这里的局域网聊天工具程序既要作为服务器端，又要作为客户端，可以将它看作P2P(端到端，peer to peer)。如果要进行聊天，首先要获取所有登录用户的信息，这是通过在每一个用户运行该程序时发送广播实现的。不仅用户登录时要进行广播，而且用户退出、发送消息、在发送文件前发送其文件名、用户拒绝接收传送来的文件等信息都要使用UDP广播来告知所有用户或者指定的用户。整个程序的示意图如图5-5所示。

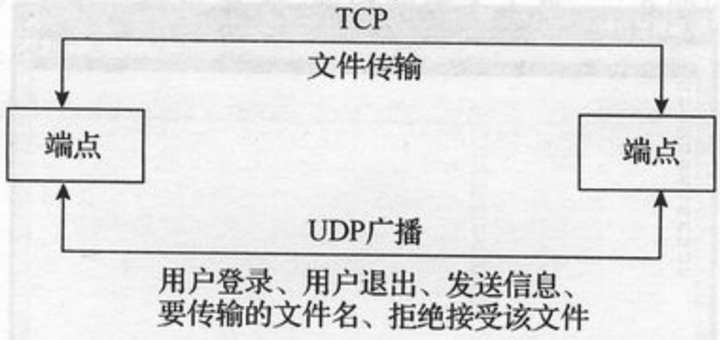



图5-5 局域网聊天工具功能实现示意图

下面就来编写代码，通过使用UDP来实现用户信息的显示和基本的聊天功能。(项目源码路径：src\5\5-2\chat)首先要在widget.h文件中定义一个枚举变量MessageType，它用来区分不同的广播类型，其值有Message、NewParticipant、ParticipantLeft、FileName和Refuse，分别用来表示消息、新用户加入、用户退出、文件名和拒绝接收文件5种不同的信息类型。然后还要定义一些函数和变量，完成后widget.h文件的内容如下：

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
class QUdpSocket;

namespace Ui {
class Widget;
}

// 枚举变量标志信息类型，分别为消息、新用户加入、用户退出、文件名、拒绝接收文件
enum MessageType{Message, NewParticipant, ParticipantLeft, FileName, Refuse};

class Widget : public QWidget
{
```

```
    Q_OBJECT

public:
    explicit Widget(QWidget * parent = 0);
    ~Widget();

protected:
    void newParticipant(QString userName,
                        QString localHostName, QString ipAddress);
    void participantLeft(QString userName,
                         QString localHostName, QString time);
    void sendMessage(MessageType type, QString serverAddress = "");

    QString getIP();
    QString getUserName();
    QString getMessage();

private:
    Ui::Widget * ui;
    QUdpSocket * udpSocket;
    qint16 port;

private slots:
    void processPendingDatagrams();
};
#endif // WIDGET_H
```

然后转到 widget.cpp 文件，先添加头文件包含：

```
#include <QUdpSocket>
#include <QHostInfo>
#include <QMessageBox>
#include <QScrollBar>
#include <QDateTime>
#include <QNetworkInterface>
#include <QProcess>
```

然后在构造函数中添加如下代码：

```
udpSocket = new QUdpSocket(this);
port = 45454;
udpSocket->bind(port, QUdpSocket::ShareAddress | QUdpSocket::ReuseAddressHint);
connect(udpSocket, SIGNAL(readyRead()), this, SLOT(processPendingDatagrams()));
sendMessage(NewParticipant);
```

这里创建了 UDP 套接字并进行了初始化，最后调用了 sendMessage()函数来广播用户登录信息，该函数用来发送各种 UDP 数据，下面添加它的定义代码：

```
void Widget::sendMessage(MessageType type, QString serverAddress)
{
  QByteArray data;
  QDataStream out(&data, QIODevice::WriteOnly);
  QString localHostName = QHostInfo::localHostName();
  QString address = getIP();
  out << type << getUserName() << localHostName;

  switch (type)
  {
  case Message :
    if (ui->messageTextEdit->toPlainText() == "") {
      QMessageBox::warning(0, tr("警告"),
                tr("发送内容不能为空"),QMessageBox::Ok);
      return;
    }
    out << address << getMessage();
    ui->messageBrowser->verticalScrollBar()
        ->setValue(ui->messageBrowser->verticalScrollBar()->maximum());
    break;

  case NewParticipant :
    out << address;
    break;

  case ParticipantLeft :
    break;

  case FileName :
    break;

  case Refuse :
    break;
  }
  udpSocket->writeDatagram(data,data.length(), QHostAddress::Broadcast, port);
}
```

在该函数中，首先向要发送的数据中写入信息类型 type、用户名（使用 getUserName()函数获取）和主机名。其中，type 用于接收端区分信息类型，从而可以对不同的信息进行不同的处理。然后对不同的信息进行了不同的操作：对于普通的聊天消息 Message，先判断发送的消息是否为空，如果为空则进行警告，然后向发送的数据中又写入了本机的 IP 地址和输入的消息文本；对于新用户加入 NewParticipant，只是简单地向数据中写入 IP 地址；对于用户离开 ParticipantLeft，不需要进行其他任何操作；而对于发送文件名 FileName 和拒绝接收文件 Refuse，这里先不进行处

理，等到后面再添加代码。完成对信息的处理后，最后使用 writeDatagram()函数进行了 UDP 广播。

该程序同时要接收 UDP 广播发送来的数据，这是通过 processPendingDatagrams()槽实现的，下面来添加它的定义：

```
void Widget::processPendingDatagrams()
{
  while (udpSocket->hasPendingDatagrams())
  {
    QByteArray datagram;
    datagram.resize(udpSocket->pendingDatagramSize());
    udpSocket->readDatagram(datagram.data(), datagram.size());
    QDataStream in(&datagram, QIODevice::ReadOnly);
    int messageType;
    in >> messageType;
    QString userName, localHostName, ipAddress, message;
    QString time = QDateTime::currentDateTime()
                        .toString("yyyy-MM-dd hh:mm:ss");
    switch (messageType)
    {
    case Message:
      in >> userName >> localHostName >> ipAddress >> message;
      ui->messageBrowser->setTextColor(Qt::blue);
      ui->messageBrowser->setCurrentFont(QFont("Times New Roman", 12));
      ui->messageBrowser->append("[ " + userName + " ]" + time);
      ui->messageBrowser->append(message);
      break;

    case NewParticipant:
      in >> userName >> localHostName >> ipAddress;
      newParticipant(userName, localHostName, ipAddress);
      break;

    case ParticipantLeft:
      in >> userName >> localHostName;
      participantLeft(userName, localHostName, time);
      break;

    case FileName:
      break;

    case Refuse:
      break;
    }
```

```
    }
}
```

每当有数据到来时都会触发该槽，这里首先获取了信息的类型。下面对不同的信息类型进行了不同的操作：如果是普通的聊天消息 Message，那么就获取其中的用户名、主机名、IP 地址和消息等数据，然后将用户名和消息显示在界面左上角的信息浏览器中；如果是新用户加入 NewParticipant，那么就获取其中的用户名、主机名和 IP 地址信息，然后使用 newParticipant()函数进行新用户登录的处理；如果是用户退出 ParticipantLeft，那么获取其中的用户名和主机名，然后使用 participantLeft()函数进行用户离开的处理；对于 FileName 和 Refuse 类型的信息，这里也先不进行实现。

下面来添加处理新用户加入的 newParticipant()函数的定义：

```
void Widget::newParticipant(QString userName, QString localHostName, QString ipAddress)
{
    bool isEmpty = ui->userTableWidget
            ->findItems(localHostName, Qt::MatchExactly).isEmpty();
    if (isEmpty) {
        QTableWidgetItem *user = new QTableWidgetItem(userName);
        QTableWidgetItem *host = new QTableWidgetItem(localHostName);
        QTableWidgetItem *ip = new QTableWidgetItem(ipAddress);

        ui->userTableWidget->insertRow(0);
        ui->userTableWidget->setItem(0, 0, user);
        ui->userTableWidget->setItem(0, 1, host);
        ui->userTableWidget->setItem(0, 2, ip);

        ui->messageBrowser->setTextColor(Qt::gray);
        ui->messageBrowser->setCurrentFont(QFont("Times New Roman",10));
        ui->messageBrowser->append(tr("%1 在线!").arg(userName));

        ui->userNumLabel->setText(tr("在线人数：%1")
                                  .arg(ui->userTableWidget->rowCount()));

        sendMessage(NewParticipant);
    }
}
```

这里先使用主机名来判断该用户是否已经加入到用户列表中，如果没有加入则向界面右侧的用户列表中添加新用户的信息，然后在信息浏览器中显示用户的加入。这里要注意，该函数的最后再次调用了 sendMessage()函数来发送新用户登录信息，这是因为已经在线的各个端点也要告知刚加入的端点它们自己的用户信息，如果不这样做，那么在新加入的端点的用户列表中就无法显示其他已经在线的端点的信息。

下面来添加处理用户离开的 participantLeft()函数的定义：

```
void Widget::participantLeft(QString userName, QString localHostName, QString time)
{
  int rowNum = ui->userTableWidget->findItems(localHostName,
                                Qt::MatchExactly).first()->row();
  ui->userTableWidget->removeRow(rowNum);
  ui->messageBrowser->setTextColor(Qt::gray);
  ui->messageBrowser->setCurrentFont(QFont("Times New Roman", 10));
  ui->messageBrowser->append(tr("%1 于 %2 离开!").arg(userName).arg(time));
  ui->userNumLabel->setText(tr("在线人数：%1")
                        .arg(ui->userTableWidget->rowCount()));
}
```

这里主要是在用户列表中将退出的用户的信息删除掉，然后进行提示。下面添加用于获取 IP 地址和用户名的函数的定义：

```
// 获取 ip 地址
QString Widget::getIP()
{
  QList<QHostAddress> list = QNetworkInterface::allAddresses();
  foreach (QHostAddress address, list) {
    if(address.protocol() == QAbstractSocket::IPv4Protocol)
      return address.toString();
  }
  return 0;
}

// 获取用户名
QString Widget::getUserName()
{
  QStringList envVariables;
  envVariables << "USERNAME.*" << "USER.*" << "USERDOMAIN.*"
               << "HOSTNAME.*" << "DOMAINNAME.*";
  QStringList environment = QProcess::systemEnvironment();
  foreach (QString string, envVariables) {
    int index = environment.indexOf(QRegExp(string));
    if (index != -1) {
      QStringList stringList = environment.at(index).split('=');
      if (stringList.size() == 2) {
        return stringList.at(1);
        break;
      }
    }
  }
```

```
    return "unknown";
}
```

用户名使用了 QProcess 类的相关函数进行获取，这里不再进行过多讲解。下面添加 getMessage()函数的定义，它用来获取用户输入的聊天消息并进行一些设置：

```
QString Widget::getMessage()
{
    QString msg = ui->messageTextEdit->toHtml();
    ui->messageTextEdit->clear();
    ui->messageTextEdit->setFocus();
    return msg;
}
```

这里从界面的消息文本编辑器中获取了用户输入的消息，然后将文本编辑器中的内容进行了清空。下面转到设计模式，然后进入“发送”按钮的单击信号 clicked()对应的槽，更改如下：

```
void Widget::on_sendButton_clicked()
{
    sendMessage(Message);
}
```

这里简单调用了 sendMessage()函数来发送消息。因为在代码中还使用了中文，所以还要在 main.cpp 文件中添加相应的代码。下面运行程序，然后在输入栏中输入一些消息并按下发送按钮，效果如图 5-6 所示。也可以在同一局域网中的不同计算机上同时运行该程序，查看是否可以进行聊天。

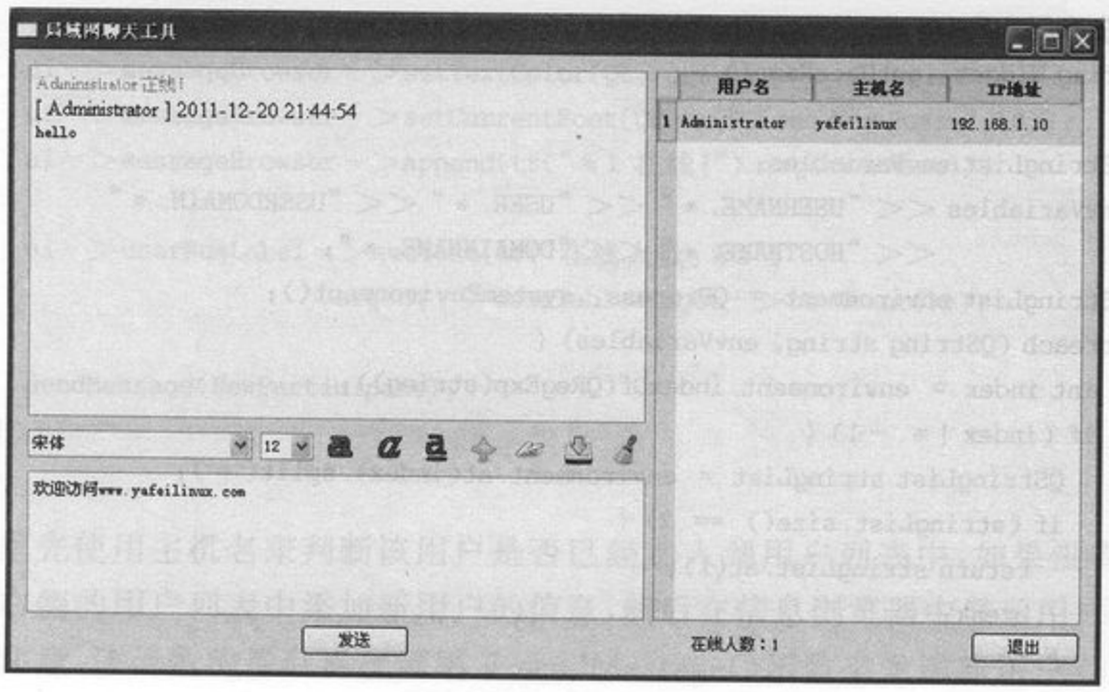



图 5-6　使用局域网聊天工具实现聊天功能

## 5.3　实现文件传输功能

文件传输使用 TCP 来实现，这里创建两个新的类来分别实现 TCP 服务器和 TCP 客户端的功能。对于文件传输的流程简单描述如下：在主界面用户列表中先选中要给其发送文件的用户，然后按下传输文件图标打开发送文件对话框。在该对话框中用户要先选择传输的文件，然后按下“发送”按钮，这时会先使用 UDP 广播将文件名发送给接收端；接收端收到了发送文件的 UDP 信息时就会弹出一个提示框，询问是否要接收指定的文件，如果同意接收，则在接收端创建 TCP 客户端，然后使用 TCP 进行文件传输。如果拒绝接收该文件，那么会使用 UDP 广播将拒绝信息发送给发送端，一旦发送端收到该信息就取消文件的传输。

### 5.3.1　创建 TCP 服务器类

在 TCP 服务器类中，要创建一个发送端对话框来选择文件并进行发送。实际的服务器是通过新创建的 QTcpServer 对象实现的。当打开文件并单击“发送”按钮后，服务器进入监听状态并使用 UDP 广播将要传输的文件名发送给接收端，如果接收端拒绝接收该文件，则关闭服务器，否则进行正常的 TCP 数据传输。

(项目源码路径：src\5\5 - 3\chat)向项目中添加新的 Qt 设计师界面类，界面模板选择 Dialog without Buttons，类名更改为 TcpServer。完成后在打开的 tcpserver.ui 中，向界面上拖入一个 Label 部件、一个 Progress Bar 部件和 3 个 Push Button 部件，最终的界面效果如图 5 - 7 所示。

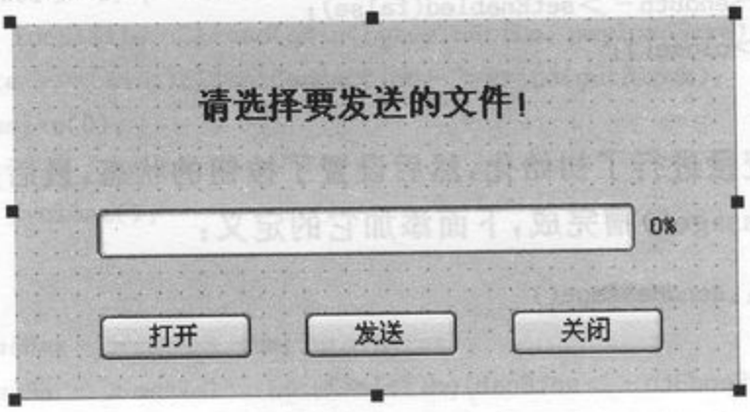



图 5 - 7　服务器端界面设计效果

先将界面的 windowTitle 属性设置为“发送端”。然后将 Label 的显示文本更改为“请选择要发送的文件!”，再更改如下属性：objectName 更改为 serverStatusLabel；在 font 中将点大小设置为 12，然后选中“粗体”；在 alignment 中将“水平的”选择为 AlignHCenter。将 Progress Bar 部件的 objectName 保持为默认的 progressBar 即可，将其 value 属性设置为 0；将“打开”按钮、“发送”按钮和“关闭”按钮的 object-

Name 分别更改为 serverOpenBtn、serverSendBtn 和 serverCloseBtn。

下面进入 tcpserver.h 文件，在其中添加变量定义和函数声明，因为篇幅原因这里就不再列出 tcpserver.h 文件的内容，读者需要下载本书的源码进行参考。下面再转到 tcpserver.cpp 文件，首先添加头文件包含，然后在构造函数中添加如下代码：

```
setFixedSize(350,180);

tcpPort = 6666;
tcpServer = new QTcpServer(this);
connect(tcpServer, SIGNAL(newConnection()), this, SLOT(sendMessage()));

initServer();
```

这里首先将对话框大小固定为 350×180，然后创建了 QTcpServer 对象并进行信号和槽的关联。最后调用了初始服务器的 initServer()函数，下面添加该函数的定义：

```
void TcpServer::initServer()
{
  payloadSize = 64 * 1024;
  TotalBytes = 0;
  bytesWritten = 0;
  bytesToWrite = 0;
  ui->serverStatusLabel->setText(tr("请选择要传送的文件"));
  ui->progressBar->reset();
  ui->serverOpenBtn->setEnabled(true);
  ui->serverSendBtn->setEnabled(false);
  tcpServer->close();
}
```

这里对一些变量进行了初始化，然后设置了按钮的状态，最后关闭了服务器。发送数据由 sendMessage()槽完成，下面添加它的定义：

```
void TcpServer::sendMessage()
{
  ui->serverSendBtn->setEnabled(false);
  clientConnection = tcpServer->nextPendingConnection();
  connect(clientConnection, SIGNAL(bytesWritten(qint64)),
    this, SLOT(updateClientProgress(qint64)));

  ui->serverStatusLabel->setText(tr("开始传送文件 %1 !").arg(theFileName));

  localFile = new QFile(fileName);
  if(! localFile->open((QFile::ReadOnly))){
    QMessageBox::warning(this, tr("应用程序"), tr("无法读取文件 %1:\n%2").arg"
```

```
"(fileName).arg(localFile->errorString()));
        return;
    }
    TotalBytes = localFile->size();
    QDataStream sendOut(&outBlock, QIODevice::WriteOnly);
    sendOut.setVersion(QDataStream::Qt_4_7);
    time.start(); // 开始计时
    QString currentFile = fileName.right(fileName.size()
                                         - fileName.lastIndexOf('/') - 1);
    sendOut << qint64(0) << qint64(0) << currentFile;
    TotalBytes += outBlock.size();
    sendOut.device()->seek(0);
    sendOut << TotalBytes << qint64((outBlock.size() - sizeof(qint64) * 2));
    bytesToWrite = TotalBytes - clientConnection->write(outBlock);
    outBlock.resize(0);
}
```

这里代码的作用我们在《Qt Creator 快速入门》中第 18 章 TCP 编程部分已经详细讲解过了。需要说明的是，这里使用了 time.start()来启动计时，time 是 QTime 对象，用来统计传输所用的时间，这是在更新进度条槽中实现的，下面添加该槽的定义：

```
void TcpServer::updateClientProgress(qint64 numBytes)
{
    qApp->processEvents();
    bytesWritten += (int)numBytes;
    if (bytesToWrite > 0) {
        outBlock = localFile->read(qMin(bytesToWrite, payloadSize));
        bytesToWrite -= (int)clientConnection->write(outBlock);
        outBlock.resize(0);
    } else {
        localFile->close();
    }

    ui->progressBar->setMaximum(TotalBytes);
    ui->progressBar->setValue(bytesWritten);

    float useTime = time.elapsed();
    double speed = bytesWritten / useTime;
    ui->serverStatusLabel->setText(tr("已发送 %1MB (%2MB/s) "
                "\n共 %3MB 已用时：%4 秒\n 估计剩余时间：%5 秒")
                .arg(bytesWritten / (1024 * 1024))
                .arg(speed * 1000 / (1024 * 1024), 0, 'f', 2)
                .arg(TotalBytes / (1024 * 1024))
                .arg(useTime/1000, 0, 'f', 0)
```

```
                .arg(TotalBytes/speed/1000 - useTime/1000, 0, 'f', 0));

    if(bytesWritten == TotalBytes) {
      localFile->close();
      tcpServer->close();
      ui->serverStatusLabel->setText(tr("传送文件 %1 成功")
                                     .arg(theFileName));
    }
  }
```

这里需要说明的是 qApp→processEvents()函数用于在传输大文件时使界面不会冻结，这个可以参考《Qt Creator 快速入门》中 3.2.3 小节 QProgressDialog 部分的讲解。代码中使用了 time.elapsed()函数来获取耗费的时间，这个是从 time.start()开始计时的。通过获取的时间在界面上显示出了传输速度等信息。

下面分别从设计模式进入“打开”按钮、“发送”按钮和“关闭”按钮的单击信号对应的槽，更改如下：

```
// 打开按钮
void TcpServer::on_serverOpenBtn_clicked()
{
  fileName = QFileDialog::getOpenFileName(this);
  if(! fileName.isEmpty())
  {
    theFileName = fileName.right(fileName.size() - fileName.lastIndexOf('/') - 1);
    ui->serverStatusLabel->setText(tr("要传送的文件为：%1 ").arg(theFileName));
    ui->serverSendBtn->setEnabled(true);
    ui->serverOpenBtn->setEnabled(false);
  }
}

// 发送按钮
void TcpServer::on_serverSendBtn_clicked()
{
  if(! tcpServer->listen(QHostAddress::Any, tcpPort)) // 开始监听
  {
    qDebug() << tcpServer->errorString();
    close();
    return;
  }
  ui->serverStatusLabel->setText(tr("等待对方接收... ..."));
  emit sendFileName(theFileName);
}

// 关闭按钮
```

```
void TcpServer::on_serverCloseBtn_clicked()
{
  if(tcpServer->isListening())
  {
    tcpServer->close();
    if (localFile->isOpen())
      localFile->close();
    clientConnection->abort();
  }
  close();
}
```

按下打开按钮后弹出一个文件对话框，选择完要发送的文件后更新了按钮的状态；在按下发送按钮后将服务器设置为监听状态然后发送了 sendFileName()信号，在主界面类中将关联该信号并使用 UDP 广播将文件名发送给接收端；按下“关闭”按钮后，先关闭服务器，然后关闭该对话框。如果接收端拒绝接收该文件，便关闭服务器，这是通过 refused()函数实现的，下面添加该函数的定义：

```
void TcpServer::refused()
{
  tcpServer->close();
  ui->serverStatusLabel->setText(tr("对方拒绝接收!!!"));
}
```

这个函数在主界面 Widget 类中当接收到接收端的拒绝接收文件的 UDP 信息时被调用。下面再添加关闭事件的处理函数，这里只是简单调用了关闭按钮单击信号槽：

```
void TcpServer::closeEvent(QCloseEvent *)
{
  on_serverCloseBtn_clicked();
}
```

### 5.3.2　创建 TCP 客户端类

前面创建了 TCP 服务器类来发送文件，这一节中再添加一个 TCP 客户端类，用于接收文件。

继续在前面的项目中更改。向项目中添加新的 Qt 设计师界面类，界面模版选择 Dialog without Buttons，类名更改为 TcpClient。完成后在打开的 tcpclient.ui 中，向界面上拖入一个 Label 部件、一个 Progress Bar 部件和两个 Push Button 部件，最终的界面效果如图 5-8 所示。

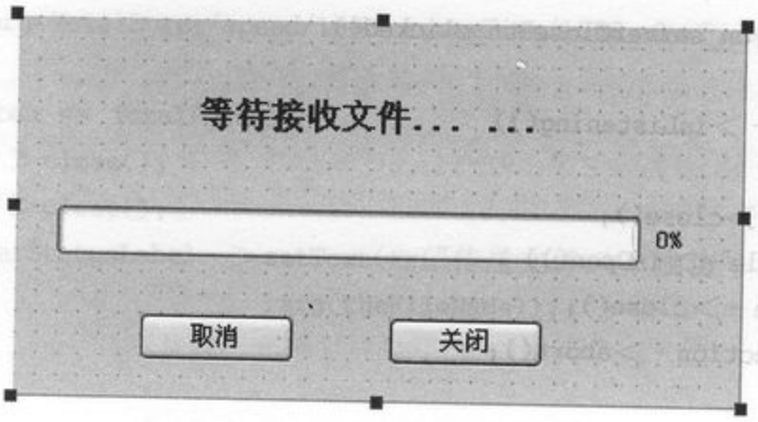



图5-8 客户端界面设计效果

先将界面的windowTitle属性设置为“接收端”。然后将Label的显示文本更改为“等待接收文件… …”，再更改如下属性：objectName更改为tcpClientStatusLabel；在font中将点大小设置为12，然后选中“粗体”；在alignment中将“水平的”选择为AlignHCenter。Progress Bar部件的objectName保持为默认的progressBar即可，将其value属性设置为0；将“取消”按钮和“关闭”按钮的objectName分别更改为tcpClientCancleBtn和tcpClientCloseBtn。

设计完界面后进入tcpclient.h文件中添加函数声明和变量定义，这里也不再列出该文件的内容。下面转入tcpclient.cpp文件，先添加头文件包含，然后在构造函数中添加如下代码：

```
setFixedSize(350,180);

TotalBytes = 0;
bytesReceived = 0;
fileNameSize = 0;
tcpClient = new QTcpSocket(this);
tcpPort = 6666;

connect(tcpClient, SIGNAL(readyRead()), this, SLOT(readMessage()));
connect(tcpClient, SIGNAL(error(QAbstractSocket::SocketError)),
  this, SLOT(displayError(QAbstractSocket::SocketError)));
```

这里对变量进行了初始化，并创建了QTcpServer对象，然后关联了信号和槽。因为要在主界面Widget类中弹出文件对话框来选择发送过来文件的保存路径，所以要在客户端类中提供函数来获取该路径，这是由setFileName()函数来完成的，下面添加该函数的定义：

```
void TcpClient::setFileName(QString fileName)
{
  localFile = new QFile(fileName);
}
```

另外还要从主界面类中获取发送端的 IP 地址，这是由 setHostAddress()函数来实现的：

```
void TcpClient::setHostAddress(QHostAddress address)
{
  hostAddress = address;
  newConnect();
}
```

这里的 newConnect()槽是用来设置与服务器的连接的，下面添加它的的定义：

```
void TcpClient::newConnect()
{
  blockSize = 0;
  tcpClient->abort();
  tcpClient->connectToHost(hostAddress, tcpPort);
  time.start();
}
```

建立好连接后便开始计时。下面添加读取数据 readMessage()槽的定义：

```
void TcpClient::readMessage()
{
  QDataStream in(tcpClient);
  in.setVersion(QDataStream::Qt_4_7);

  float useTime = time.elapsed();

  if (bytesReceived <= sizeof(qint64) * 2) {
    if ((tcpClient->bytesAvailable()>= sizeof(qint64) * 2) && (fileNameSize == 0))
    {
      n >> TotalBytes >> fileNameSize;
      bytesReceived += sizeof(qint64) * 2;
    }
    if((tcpClient->bytesAvailable() >= fileNameSize) && (fileNameSize != 0)){
      in>>fileName;
      bytesReceived += fileNameSize;
      if (! localFile->open(QFile::WriteOnly)) {
    QMessageBox::warning(this, tr("应用程序"),tr("无法读取文件 %1:\n%2.")
                .arg(fileName).arg(localFile->errorString()));
        return;
      }
    } else {
      return;
    }
  }
  if (bytesReceived < TotalBytes) {
```

```
        bytesReceived += tcpClient->bytesAvailable();
        inBlock = tcpClient->readAll();
        localFile->write(inBlock);
        inBlock.resize(0);
    }

    ui->progressBar->setMaximum(TotalBytes);
    ui->progressBar->setValue(bytesReceived);

    double speed = bytesReceived / useTime;
    ui->tcpClientStatusLabel->setText(tr("已接收 %1MB (%2MB/s) "
                "\n共%3MB 已用时:%4 秒\n估计剩余时间:%5 秒")
                .arg(bytesReceived / (1024 * 1024))
                .arg(speed * 1000/(1024 * 1024),0,f,2)
                .arg(TotalBytes / (1024 * 1024))
                .arg(useTime/1000,0,f,0)
                .arg(TotalBytes/speed/1000 - useTime/1000,0,f,0));

    if (bytesReceived == TotalBytes) {
        localFile->close();
        tcpClient->close();
        ui->tcpClientStatusLabel->setText(tr("接收文件 %1 完毕").arg(fileName));
    }
}
```

这里的代码也已经在《Qt Creator 快速入门》中第 18 章详细讲解过了。下面添加 displayError()槽的实现:

```
void TcpClient::displayError(QAbstractSocket::SocketError socketError)
{
    switch(socketError)
    {
    case QAbstractSocket::RemoteHostClosedError : break;
    default : qDebug() << tcpClient->errorString();
    }
}
```

下面进入设计模式,分别进入“取消”按钮和“关闭”按钮的单击信号槽中,更改如下:

```
// 取消按钮
void TcpClient::on_tcpClientCancleBtn_clicked()
{
    tcpClient->abort();
    if (localFile->isOpen())
        localFile->close();
```

```
}
// 关闭按钮
void TcpClient::on_tcpClientCloseBtn_clicked()
{
  tcpClient->abort();
  if (localFile->isOpen())
    localFile->close();
  close();
}
```

最后添加关闭事件处理函数的定义：

```
void TcpClient::closeEvent(QCloseEvent *)
{
  on_tcpClientCloseBtn_clicked();
}
```

### 5.3.3　在主界面中进行文件的发送和接收

下面通过在主界面中添加代码来实现文件的发送和接收。首先在 widget.h 文件中添加类的前置声明：

```
class TcpServer;
```

然后添加一个 protected 函数声明：

```
void hasPendingFile(QString userName, QString serverAddress,
                    QString clientAddress, QString fileName);
```

该函数用于在收到文件名 UDP 信息时判断是否接收该文件。下面再添加 private 变量和对象的定义：

```
QString fileName;
TcpServer *server;
```

再添加一个私有槽声明：

```
void getFileName(QString);
```

它用来获取服务器类 sendFileName()信号发送过来的文件名。下面转到 widget.cpp 文件中，先添加头文件包含：

```
#include "tcpserver.h"
#include "tcpclient.h"
#include <QFileDialog>
```

然后在构造函数中继续添加如下代码：

```
server = new TcpServer(this);
connect(server, SIGNAL(sendFileName(QString)), this,SLOT(getFileName(QString)));
```

这里创建了服务器类对象,并且关联了其中的sendFileName()信号。下面添加getFileName()槽的定义:

```
void Widget::getFileName(QString name)
{
  fileName = name;
  sendMessage(FileName);
}
```

这里获取了文件名,然后发送FileName类型的UDP广播。下面从设计模式转入“传输文件”按钮单击信号对应的槽,更改如下:

```
void Widget::on_sendToolBtn_clicked()
{
  if (ui->userTableWidget->selectedItems().isEmpty())
  {
    QMessageBox::warning(0, tr("选择用户"),
                         tr("请先从用户列表选择要传送的用户!"), QMessageBox::Ok);
    return;
  }
  server->show();
  server->initServer();
}
```

这里必须先在用户列表中选择一个用户来接收文件,然后弹出发送端界面。下面将sendMessage()中的FileName和Refuse处的代码更改为:

```
case FileName : {
    int row = ui->userTableWidget->currentRow();
    QString clientAddress = ui->userTableWidget->item(row, 2)->text();
    out << address << clientAddress << fileName;
    break;
}
case Refuse :
    out << serverAddress;
    break;
```

因为在文件传输以前,先要选择该文件要发送给的用户。这里就是从列表中获取了当前选中用户的IP地址,然后将其与文件名一起写入UDP数据报中。下面再将接收UDP信息的槽processPendingDatagrams()中的FileName和Refuse处的代码更改为:

```
case FileName: {
```

```
    in >> userName >> localHostName >> ipAddress;
    QString clientAddress, fileName;
    in >> clientAddress >> fileName;
    hasPendingFile(userName, ipAddress, clientAddress, fileName);
    break;
  }

  case Refuse: {
    in >> userName >> localHostName;
    QString serverAddress;
    in >> serverAddress;
    QString ipAddress = getIP();
    if (ipAddress == serverAddress)
    {
      server->refused();
    }
    break;
  }
```

发送过来文件名信息时使用了 hasPendingFile()函数来判断是否要接收该文件;当发送过来拒绝信息时,要判断该程序是否是发送端,如果是则执行服务器的 refused()函数。下面添加 hasPendingFile()函数的定义:

```
void Widget::hasPendingFile(QString userName, QString serverAddress,
                            QString clientAddress, QString fileName)
{
  QString ipAddress = getIP();
  if(ipAddress == clientAddress)
  {
    int btn = QMessageBox::information(this, tr("接受文件"),
                tr("来自 %1(%2)的文件:%3,是否接收?")
              .arg(userName).arg(serverAddress).arg(fileName),
                QMessageBox::Yes, QMessageBox::No);
    if (btn == QMessageBox::Yes) {
      QString name = QFileDialog::getSaveFileName(0,tr("保存文件"), fileName);
      if(! name.isEmpty())
      {
        TcpClient *client = new TcpClient(this);
        client->setFileName(name);
        client->setHostAddress(QHostAddress(serverAddress));
        client->show();
      }
    } else {
      sendMessage(Refuse, serverAddress);
    }
```

```
    }
}
```

在该函数中先弹出一个提示框让用户判断是否要接收发送过来的文件，如果接收，那么就创建客户端对象来传输文件；如果拒绝接收，那么就发送拒绝信息的 UDP 广播。

到这里文件传输功能就基本上完成了，读者可以运行程序然后进行文件的传输。比如自己给自己传输文件，效果如图 5-9 所示。

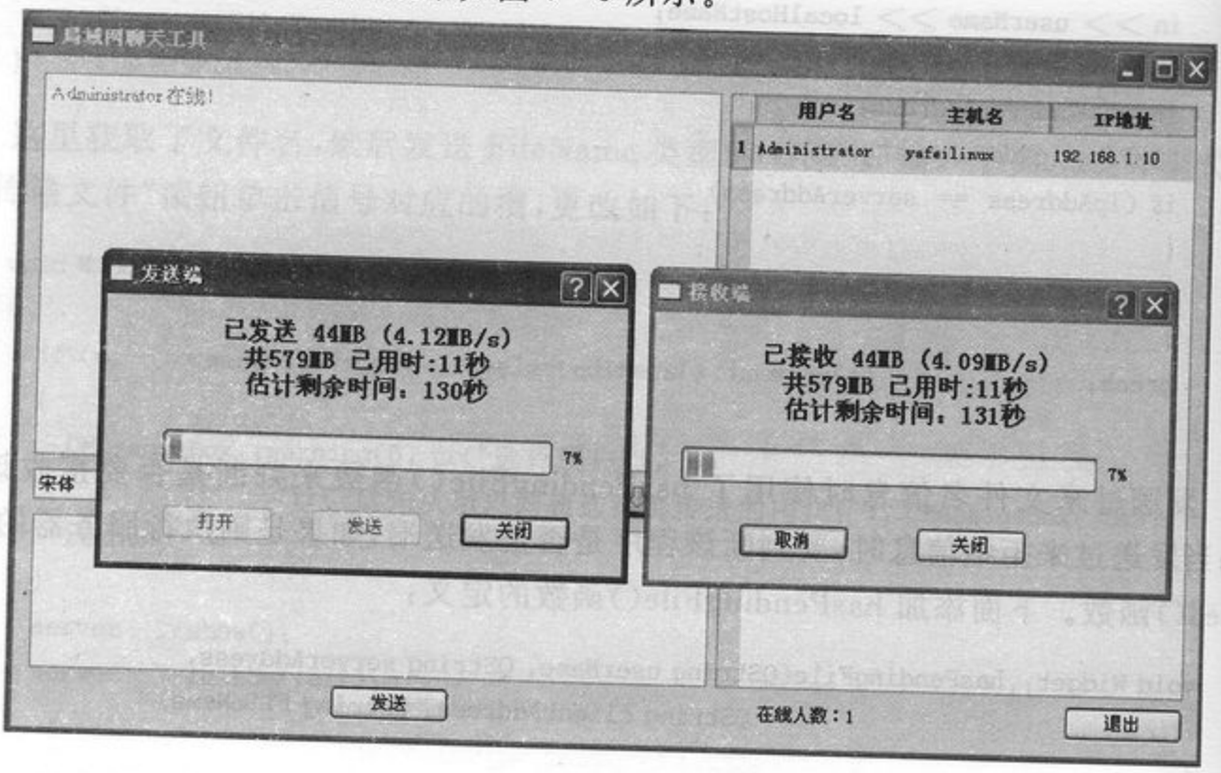



图 5-9　传输文件运行效果

## 5.4　完善程序功能

### 5.4.1　更改字体

(项目源码路径：src\5\5-4\chat)下面来实现更改字体的功能，主要是更改聊天时文本的字体、大小、颜色等。

**1. 更改字体族**

使用界面上的 Font Combo Box 部件来更改聊天文本的字体族。首先双击 widget.ui 文件进入设计模式，然后在界面上的 Font Combo Box 部件上右击，选择“转到槽”，然后选择 currentFontChanged(QFont)信号并按下“确定”按钮。完成后在该信号对应的槽中添加如下代码：

```
void Widget::on_fontComboBox_currentFontChanged(QFont f)
```

```
{
  ui->messageTextEdit->setCurrentFont(f);
  ui->messageTextEdit->setFocus();
}
```

这里获取了当前选择的字体，然后在消息文本编辑器中使用该字体。

**2. 更改字体大小**

从设计模式进入界面上 Combo Box 部件的 currentIndexChanged(QString)信号对应的槽，更改如下：

```
void Widget::on_sizeComboBox_currentIndexChanged(QString size)
{
  ui->messageTextEdit->setFontPointSize(size.toDouble());
  ui->messageTextEdit->setFocus();
}
```

**3. 设置字体加粗、倾斜、下划线和颜色**

先分别进入 boldToolBtn、italicToolBtn 和 underlineToolBtn 的 clicked(bool)信号对应的槽，更改如下：

```
// 加粗
void Widget::on_boldToolBtn_clicked(bool checked)
{
  if(checked)
    ui->messageTextEdit->setFontWeight(QFont::Bold);
  else
    ui->messageTextEdit->setFontWeight(QFont::Normal);
  ui->messageTextEdit->setFocus();
}

// 倾斜
void Widget::on_italicToolBtn_clicked(bool checked)
{
  ui->messageTextEdit->setFontItalic(checked);
  ui->messageTextEdit->setFocus();
}

// 下划线
void Widget::on_underlineToolBtn_clicked(bool checked)
{
  ui->messageTextEdit->setFontUnderline(checked);
  ui->messageTextEdit->setFocus();
}
```

然后进入 colorToolBtn 的 clicked()信号对应的槽，更改如下：

```
void Widget::on_colorToolBtn_clicked()
{
  color = QColorDialog::getColor(color, this);
  if (color.isValid()) {
    ui->messageTextEdit->setTextColor(color);
    ui->messageTextEdit->setFocus();
  }
}
```

这里还需要在 widget.h 文件中定义一个私有的 QColor 对象 color，然后在 widget.cpp 中添加 #include <QColorDialog> 头文件包含。现在运行程序，更改输入的消息文本的字体和颜色等设置，效果如图 5-10 所示。

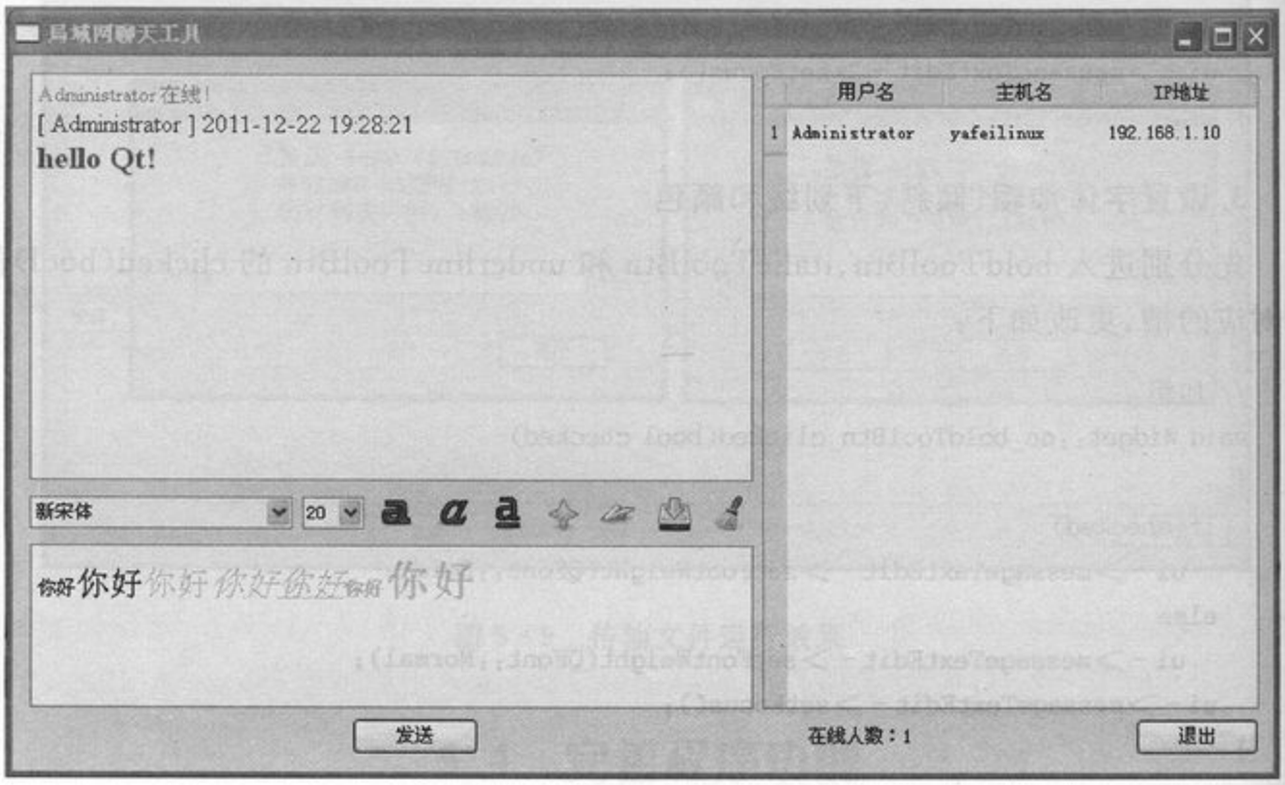



图 5-10　更改聊天文本字体效果

如果在文本编辑器中不同的几段文本使用了不同的字体格式，那么还需要添加函数来使光标在不同格式的文本上单击时可以使编辑器自动切换成对应的格式。例如在图 5-10 中几个“你好”分别使用了不同的格式设置，那么我们希望当单击到每种格式的文本时可以使设置字体的部件切换到相应格式对应的状态。

首先在 widget.h 文件中添加头文件包含：

```
#include <QTextCharFormat>
```

然后添加一个私有槽的定义：

```
void currentFormatChanged(const QTextCharFormat &format);
```

然后到 widget.cpp 文件中，在构造函数中添加信号和槽的关联：

```
connect(ui->messageTextEdit, SIGNAL(currentCharFormatChanged(QTextCharFormat)),
    this, SLOT(currentFormatChanged(const QTextCharFormat)));
```

最后添加 currentFormatChanged()槽的定义：

```
void Widget::currentFormatChanged(const QTextCharFormat &format)
{
  ui->fontComboBox->setCurrentFont(format.font());
  // 如果字体大小出错(因为我们最小的字体为 9),使用 12
  if (format.fontPointSize() < 9) {
    ui->sizeComboBox->setCurrentIndex(3);
  } else {
    ui->sizeComboBox->setCurrentIndex( ui->sizeComboBox
          ->findText(QString::number(format.fontPointSize())));
  }
  ui->boldToolBtn->setChecked(format.font().bold());
  ui->italicToolBtn->setChecked(format.font().italic());
  ui->underlineToolBtn->setChecked(format.font().underline());
  color = format.foreground().color();
}
```

## 5.4.2　保存聊天记录及其他功能

### 1. 保存聊天记录

首先在 widget.h 文件中添加 protected 函数声明：

```
bool saveFile(const QString& fileName);
```

然后从设计模式进入 saveToolBtn 的 clicked()信号对应的槽，更改如下：

```
void Widget::on_saveToolBtn_clicked()
{
  if (ui->messageBrowser->document()->isEmpty()) {
    QMessageBox::warning(0, tr("警告"),
              tr("聊天记录为空,无法保存!"), QMessageBox::Ok);
  } else {
    QString fileName = QFileDialog::getSaveFileName(this,
    tr("保存聊天记录"), tr("聊天记录"), tr("文本(*.txt);;All File(*.*)"));
    if(! fileName.isEmpty())
      saveFile(fileName);
  }
}
```

下面添加 saveFile()函数的定义：

```
bool Widget::saveFile(const QString &fileName)
```

```
{
    QFile file(fileName);
    if (! file.open(QFile::WriteOnly | QFile::Text)) {
        QMessageBox::warning(this, tr("保存文件"),
                             tr("无法保存文件 %1:\n %2").arg(fileName)
                             .arg(file.errorString()));
        return false;
    }
    QTextStream out(&file);
    out << ui->messageBrowser->toPlainText();
    return true;
}
```

**2. 清空聊天记录**

从设计模式进入 clearToolBtn 的 clicked()信号对应的槽，更改如下：

```
void Widget::on_clearToolBtn_clicked()
{
    ui->messageBrowser->clear();
}
```

**3. 退出按钮**

从设计模式进入“退出”按钮的 clicked()信号对应的槽，更改如下：

```
void Widget::on_exitButton_clicked()
{
    close();
}
```

**4. 关闭事件**

首先在 widget.h 文件中添加 protected 函数声明：

```
void closeEvent(QCloseEvent *);
```

然后在 widget.cpp 文件中添加该函数的定义：

```
void Widget::closeEvent(QCloseEvent *e)
{
    sendMessage(ParticipantLeft);
    QWidget::closeEvent(e);
}
```

在关闭程序时发送用户离开的广播，让其他端点在其用户列表中删除该用户。

到这里整个程序就设计完成了，读者可以运行程序进行各种操作。如果有多台相连的计算机，可以在其他计算机上同时运行该程序，然后测试局域网聊天和传输文件的实际效果。

## 5.5 小　结

本章讲解了一个实用的局域网聊天工具，兼具了聊天和传输文件的功能。学习本实例程序，主要是要学会怎样在实际的应用中使用 UDP 和 TCP 编程。通过对本章的学习，读者应该学会灵活应用 TCP 编程，根据自己的需要来使用 TCP 服务器类作为发送端或者接收端；也要学会在其他网络编程中灵活使用 UDP 广播来实现一些功能。

# 第二部分

# Qt Quick

# 第 6 章　Qt Quick

Qt Quick 是一些新 UI 技术的集合，用来帮助开发者创建一种现在越来越多用于手机、多媒体播放器、机顶盒以及其他便携式设备上的直观的、现代的、流畅的用户界面。简单来说，Qt Quick 是一种高级用户界面技术，使用它可以轻松地创建供移动和嵌入式设备使用的动态触摸式界面和轻量级应用程序。Qt Quick 主要由 3 部分组成：一个改进的 Qt Creator IDE（其中包含了 Qt Quick 设计器）、新增的简单易学的 QML 语言和新加入 Qt 库中名为 QtDeclarative 的模块。Qt Quick 的中文主页网址是：http://qt.nokia.com/qtquick-cn/。

在 Qt Quick 的世界里，用户界面以及它们的行为使用 QML 来描述。QML 是对 JavaScript 的一种扩展，可以让开发者和设计者利用一个声明性语法来使用 QML 元素指定每一个用户界面。这些元素是一个图形和行为的构建模块的集合，可以组合在一起形成各种组件，可以是一个简单的按钮，也可以是一个支持网络的完整的应用程序。QML 完善了 JavaScript 和 Qt 现有的基于 QObject 的类型系统间的整合，在语言级别上添加了对自动属性绑定和透明网络传输的支持。

Qt 的 QtDeclarative 模块实现了 QML 语言和对它适用的元素之间的接口，该模块还提供了一个 C++接口，可以用来在 Qt/C++应用程序中加载 QML 文件并与之通信。

需要着重指出的是，Qt Quick 是建立在 Qt 现有的框架基础之上的，QML 可以用来扩展现有的应用程序，也可以创建全新的应用程序。QML 通过 QtDeclarative 模块便可以完全支持从 C++进行扩展。对应本章的内容，可以在 Qt 帮助中查看 Qt Quick 关键字。

## 6.1　初识 QML

QML（Qt Meta-Object Language，Qt 元对象语言）是一个用来描述应用程序的用户界面的声明式语言。在 QML 中，一个用户界面被指定为一个拥有属性的对象树。这里各种各样的对象被通称为元素。因为 JavaScript 被用作 QML 的脚本语言，所以如果想深入学习 QML，是需要有一定的 JavaScript 基础的（https://developer.mozilla.org/en/JavaScript/Guide 网页中提供了相关的教程）。如果对 HTML 和 CSS 等网页技术有一定的了解也是很有帮助的，不过这不是必须的。对应本节的内容，可以在 Qt 帮助中查看 Introduction to the QML language 关键字。

在深入学习 QML 以前，先来看一个 Qt 自带的 Qt Quick 示例程序。进入 Qt Creator 的欢迎模式，然后单击“入门”子页面。这里选择“探索 Qt Quick 示例”中 Animation 分类下的 Basics 程序，打开工程后如图 6-1 所示。这里以.qml 后缀结尾的文件便是 QML 文件，以.qmlproject 结尾的文件是 QML 项目文件。

下面先双击项目列表中的 property-animation.qml 文件，然后运行程序，这时的运行效果如图 6-2 所示。下面再双击列表中的 color-animation.qml 文件，然后

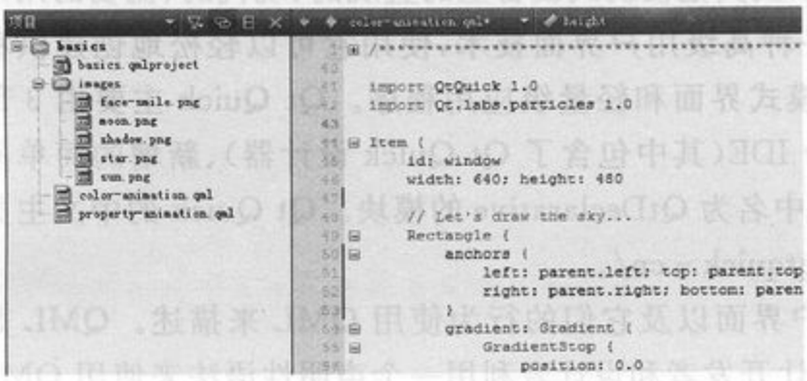



图 6-1　QML 项目编辑界面

图 6-2　QML 程序运行效果

运行程序，这时的运行效果如图 6-3 所示。可以看到，在编辑不同的.qml 文件的时候，它们是可以单独运行的。另外，QML 文件是可以直接运行而不需要进行编译的。运行 QML 文件的是一个被称为 qmlviewer.exe 的程序，它在 Qt 安装目录下的 bin 目录中。现在到该目录中找到该程序，然后运行它，效果如图 6-4 所示。可以使用该程序中的 File→Open 菜单项来打开一个 QML 文件。其实也可以在任何的文本编辑器中写入 QML 代码，并保存为.qml 格式，然后便可以使用该程序来运行了。这个程序还提供了比较实用的屏幕快照和屏幕录像功能，它们在 Recording 菜单中。不过屏幕录像并不是保存一段视频，而是保存一帧一帧的图片，注意：保存视频录像的路径中不能出现中文。在 Debugging 菜单中，可以减慢动画的速度，还可以显示错误信息。在 settings 菜单中可以进行一些设置。

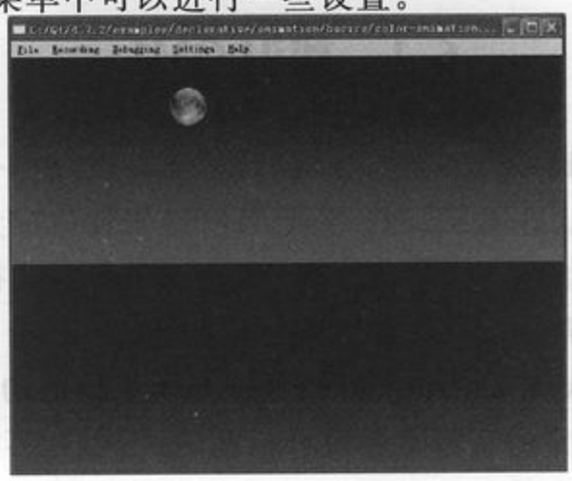



图 6-3　QML 颜色动画示例程序运行效果


图 6-4　程序 qmlviewer 运行效果

### 6.1.1　QML 基本语法

QML 的代码一般是这样的：

```
import QtQuick 1.0
Rectangle {
  width: 200
  height: 200
  color: "blue"

  Image {
    source: "pics/logo.png"
    anchors.centerIn: parent
  }
}
```

**1. 对象和属性**

代码中创建了两个对象，一个 Rectangle 对象和它的一个 Image 类型的子对象。对象由它们的类型指定，并且以大写字母开头，后面跟随一对大括号。在括号中可以添加一些附加数据，比如它的属性值和任意的子对象。比如这里 Rectangle 对象添加了 width、height 和 color 属性的值，还添加了 Image 子对象。

属性通过“属性:值”语法来指定。比如在代码中，Image 对象有一个 source 属性，被指定了“pics/logo.png”值。属性和它的值使用一个冒号隔开。属性可以分行写，例如：

```
Rectangle {
  width: 100
  height: 100
}
```

也可以将多个属性写在一行，例如：

```
Rectangle { width: 100; height: 100 }
```

当多个“属性:值”被指定在一行时，它们之间必须使用一个分号隔开。代码中的import语句导入了QtQuick模块，它包含了所有的标准QML元素。如果不使用这个import语句，那么Rectangle和Image元素就无法使用。Image中的anchors.centerIn起到了布局的作用，它会使Image处于一个对象的中心位置，比如这里就是处于其parent父对象即Rectangle的中心。

**2. 注释**

在QML中的注释和JavaScript中的注释是相似的：

- 单行注释使用“//”开始，在行的末尾结束；
- 多行注释使用“/*”开始，使用“*/”结尾。

其实，QML中的注释与C和C++中的注释的用法也是一样的，例如：

```
Text {
  text: "Hello world!" //要显示的文本
  /*
    设定文本的字体，
    可以通过设置 font 属性来完成
  */
  font.family: "Helvetica"
  font.pointSize: 24
  //opacity: 0.5
}
```

**3. 对象标识符**

每一个对象都可以指定一个唯一的id值，这样便可以在其他对象中识别并引用该对象。例如，下面的代码中有两个Text对象，第一个Text对象的id值为“text1”。现在第二个Text对象便可以引用text1.text来设置自己的text属性与第一个Text对象的text属性具有相同的值：

```
Row {
  Text {
    id: text1
    text: "Hello World"
  }

  Text { text: text1.text }
}
```

可以在一个对象所在的组件(component，会在后面的内容中讲到)中的任何地方，通过使用该对象的id来引用该对象。因此，id值在一个单一的组件中必须是唯

一的。对于一个 QML 对象而言，id 值是一个特殊的值，不要把它看作一个普通的对象属性。例如，无法使用 text1.id 来进行访问。一旦一个对象被创建，它的 id 就无法被改变了。

注意：id 值必须使用小写字母或者下划线开头，而且不能使用字母、数字和下划线以外的字符。

**4. 表达式**

JavaScript 表达式可以用于分配属性的值，例如：

```
Item {
  width: 100 * 3
  height: 50 + 22
}
```

在这些表达式中可以包含其他的对象和属性的引用，这样便创建了一个绑定：当表达式的值改变时，该表达式被分配给的属性的值会自动更新为新的值。例如：

```
Item {
  width: 300
  height: 300

  Rectangle {
    width: parent.width - 50
    height: 100
    color: "yellow"
  }
}
```

这里 Rectangle 对象的 width 属性被设置为与它的父对象的 width 相关，无论何时父对象的 width 改变了，Rectangle 的 width 都会自动更新。

## 6.1.2　编写 QML 的 Hello World 程序

当对 QML 的基本语法有了初步的了解以后，通过创建 Qt Quick 项目先来完成一个 Hello World 程序，然后再通过例子讲解后面的内容。

（项目源码路径：src\6\6 - 1\helloworld）新建 Qt Quick 项目，然后选择 Qt Quick UI，项目名称设置为 helloworld。完成后可以看到已经自动生成了一个 QML 项目文件和一个 QML 文件，并且在 helloworld.qml 文件中默认生成了一些代码。下面将 helloworld.qml 文件中的代码先更改为：

```
import QtQuick 1.0

Rectangle {
  id: myRectangle
```

```
    width: 360; height: 360
    color: "lightgray"

    Text { text: "Hello world" }
}
```

首先，需要导入所需要的类型，大多数QML文件需要导入内建的QML类型（例如Rectangle、Image等），这可以使用下面一行代码来完成：

```
import QtQuick 1.0
```

然后声明了一个Rectangle类型的根元素，最外层的元素被称为根元素。这里指定了它的id，这样可以在其他地方引用它。然后设置了它的宽width、高height、颜色color属性，宽和高的单位都是像素。Rectangle还有很多其他的属性（如x、y等），这里都使用了它们的默认值。

这里添加了一个Text元素来作为Rectangle根元素的孩子，用来显示“Hello world”文本，这是通过设置它的text属性来完成的。现在运行程序，效果如图6-5所示。

下面将Text元素的代码更改如下：

```
Text {
  text: "<h2>Hello World</h2>"; color: "darkgreen"
  x: 100; y:100
}
```

这里设置了文本的位置，这是通过指定x、y属性的值来实现的，这个位置是在父对象中的位置，这里就是在Rectangle中的位置，单位是像素。这里对text属性的值使用了HTML标签来更改字体的大小。对于颜色，代码中使用了颜色的SVG名称来进行指定，它也可以使用RGB的十六进制字符串来指定，例如：

```
color: "#002288"
```

下面运行程序，效果如图6-6所示。另外，在Qt Creator编辑器中还为编辑QML代码提供了一个Qt Quick工具栏，它会根据不同的元素显示不同的工具栏。只要在编写代码时按下“Ctrl+Alt+空格”组合键就可以调出该工具栏，当然，也可以在要插入代码处右击，然后选择“显示Qt Quick工具栏”菜单。例如，在Rectangle元素中弹出Qt Quick工具栏如图6-7所示，它提供了渐变填充、颜色和边框的设置；而在Text元素中弹出的工具栏，则是进行Text相关属性的设置，如图6-8所示。可以在工具栏上进行选择，这时便会自动在元素中添加相应的代码。

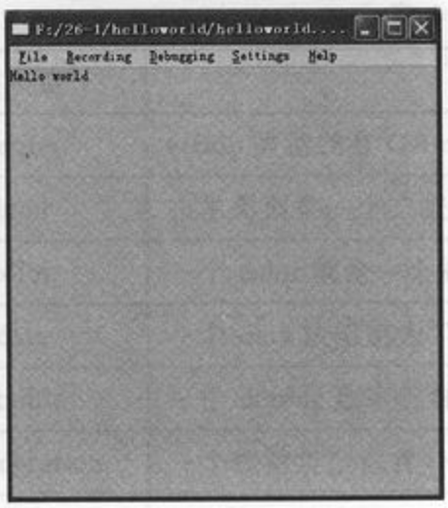



图 6-5　helloworld 程序运行效果


图 6-6　更改文本位置和颜色运行效果

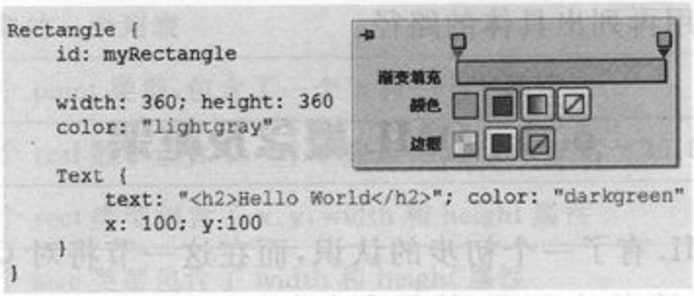



图 6-7　Rectangle 元素的工具栏


图 6-8　Text 元素的工具栏

下面来看一下 QML 的项目文件。双击 helloworld. qmlproject 文件，这时显示文档中的中文可能为乱码，这是因为 QML 程序中默认使用的是 UTF-8 编码。可以选择“编辑→选择编码”菜单项，在打开的对话框中选择 UTF-8，然后按下“根据编码重新载入”按钮。现在已经可以正常显示了，其内容如下：

```
/* Qt Creator 创建的文件 */

import QmlProject 1.0

Project {
  /* 在当前目录和子目录下包含的 qml,js 和图片文件 */
  QmlFiles {
```

```
        directory: "."
    }
    JavaScriptFiles {
        directory: "."
    }
    ImageFiles {
        directory: "."
    }
    /* 列出QML运行环境下的插件目录 */
    // importPaths: [ "../exampleplugin" ]
}
```

这里主要是指定了项目中所用的QML文件、JavaScript文件和图片文件所在的目录(默认目录为当前目录,即项目目录),指定以后,在代码中就可以直接使用该目录中的文件了,而不用再列出具体的路径。

## 6.2 QML概念及框架

前面已经对QML有了一个初步的认识,而在这一节将对QML中的各个概念展开讲解。虽然现在读者对QML的很多内容都不是很了解,但是在讲解一些内容的时候,不可避免涉及一些前面还没有讲到的知识,这时读者不要刻意去查看不懂的知识,只需要继续往下学习即可。等所有知识都学习完了,再回过头来重新学习这一节的相关内容。再次强调一下,因为这一节中会讲述QML相关的理论性知识,可能很难理解,但是它们却是QML的精华内容。也可以先跳过这一节,等有了一定的QML编程经验后,对于不太明白的用法和概念,再来本章进行选择性学习。

### 6.2.1 QML属性

#### 1. 基本的属性类型

QML支持多种类型的属性,如表6-1所列,也可以在Qt帮助中查看QML Basic Types关键字,然后分别单击各个类型查看它们具体的描述。基本的类型包括int、real、bool、string和color,例如:

```
Item {
    x: 10.5              // a real property
    state: "details"     // a string property
    focus: true          // a bool property
    ...
}
```

表 6-1 QML 基本类型

| 类 型 | 描 述 |
| --- | --- |
| action | action 类型拥有 QAction 的所有属性 |
| bool | 布尔类型是一个二进制值，有 true 和 false 两个值 |
| color | 一个 color 就是一个标准的颜色名称加上引号，例如“red” |
| date | 一个 date 被指定为“YYYY－MM－DD” |
| double | 一个 double 数字包含一个小数点，并使用双精度进行存储 |
| enumeration | 一个枚举类型包含一组已命名的值 |
| font | font 类型包含了 QFont 的属性 |
| int | 整型包含所有的数字，例如 0，10，或者－20 |
| list | 对象的一个列表 |
| point | 一个 point 类型，包含了一个 x 和一个 y 属性 |
| real | 一个 real 数字包含了一个小数点，例如 1.2 或者－29.8 |
| rect | 一个 rect 类型包含了 x、y，width 和 height 属性 |
| size | 一个 size 类型包含了 width 和 height 属性 |
| string | 一个 string 是一个在引号中的任意格式的文本，例如“Hello world!” |
| time | 一个 time 被指定为“hh：mm：ss” |
| url | 一个 URL 是一个资源定位器，像一个文件名 |
| variant | 一个 variant 类型是一个通用的属性类型 |
| vector3d | 一个 vector3d 类型包含 x、y 和 z 属性 |

QML 属性是“类型安全”的。也就是说，它们只允许分配一个匹配该属性类型的值。例如，Item 对象的 x 属性是 real 类型的，如果想为其分配一个 string 类型的值是不可以的。注意：除了后面要讲到的附加属性（Attached Properties）以外，属性总是以小写字母开头。

**2. 属性更改通知**

当一个属性更改值时，它会发送一个信号来告知这个更改。要获取这个信号，只需要创建一个信号处理器（signal handler，会在后面的内容中讲到），它使用“on＜Property＞Changed”语法来命名。例如，Rectangle 元素拥有 width 和 color 属性，下面的代码中我们在一个 Rectangle 对象中定义了两个信号处理器 onWidthChanged 和 onColorChanged，无论何时属性被修改了，都会自动调用它们：

```
Rectangle {
  width: 100; height: 100
```

```
    onWidthChanged: console.log("Width has changed to:", width)
    onColorChanged: console.log("Color has changed to:", color)
}
```

这里使用了console.log()来输出调试信息，相似的还有console.debug()，它类似于进行Qt C++编程时使用的qDebug()函数。在进行QML编程时，主要就是通过使用这两个函数来调试和信息输出的。

**3. 列表属性**

列表属性一般是这样的：

```
Item {
  children: [
    Image {},
    Text {}
  ]
}
```

列表被包含在一对方括号中，使用逗号来分隔列表元素。如果列表中只有一个元素，那么可以省去方括号：

```
Image {
  children: Rectangle {}
}
```

在列表中的条目可以使用索引来访问，这使用了“listName[index]”语法，例如：

```
Item {
  children: [
    Item { id: child1 },
    Rectangle { id: child2; width: 200 },
    Text { id: child3 }
  ]

  Component.onCompleted: {
    console.log("Width of child rectangle:", children[1].width)
  }
}
```

这里child1、child2和child3会以它们出现的顺序添加到children列表中。关于更多list基本类型的内容，可以在Qt帮助中参考QML Basic Typy关键字，然后单击list类型进行查看。这里的Component.onCompleted会在组件创建完成时执行。

**4. 默认属性**

每一个对象类型都可以指定它的列表中的一项或者一个对象属性作为它的默认属性。如果一个属性已经被声明为默认属性，那么这个属性的标签可以被省略。

例如：

```
State {
  changes: [
    PropertyChanges {},
    PropertyChanges {}
  ]
}
```

因为 changes 是 State 类型的默认属性，所以上面的代码可以简写为：

```
State {
  PropertyChanges {}
  PropertyChanges {}
}
```

**5. 分组属性**

在某些情况下，属性可以形成一个逻辑组，使用“.”或者分组符号来表示，例如：

```
Text {
  font.pixelSize: 12
  font.bold: true
}
```

也可以写成：

```
Text {
  font { pixelSize: 12; bold: true }
}
```

因为 pixelSize 和 bold 都是对字体进行设置的，所以它们可以分为一组。

**6. 附加属性**

一些对象附加属性到其他对象上。附加属性使用“Type. property”格式，其中 Type 是附加该属性的元素的类型。例如，ListView 元素会附加 ListView. isCurrentItem 属性到它的每个 delegate(委托)上：

```
Component {
  id: myDelegate
  Text {
    text: "Hello"
    color: ListView.isCurrentItem ? "red" : "blue"
  }
}
ListView {
  delegate: myDelegate
}
```

另外一个例子是Keys元素附加属性到任何可见的Item上来处理键盘按下：

```
Item {
  focus: true
  Keys.onSelectPressed: console.log("Selected")
}
```

### 6.2.2 属性绑定

属性绑定是指定一个属性的值的声明式表示方法。通过绑定可以使用一个JavaScript表达式来作为属性的值，其中可以使用其他属性的值，也可以使用在应用程序中可以获取的其他的值。如果其他属性或者数据的值更改了，这个属性的值会自动更新。每当为一个属性分配了一个JavaScript表达式时，便会隐含的使用属性绑定。对应本节内容，可以在Qt帮助中查看Property Binding关键字。

下面的QML代码片段使用了两个属性绑定将Rectangle的大小与otherItem的大小进行了关联：

```
Rectangle {
  width: otherItem.width
  height: otherItem.height
}
```

QML扩展了标准的JavaScript引擎，所以任何可用的JavaScript表达式都可以用作属性绑定。在绑定中可以访问对象属性，调用函数，甚至可以使用Date和Math等内建的JavaScript对象。向一个属性分配一个常量值也会被认为是一个绑定，因为一个常量也是一个可用的JavaScript表达式。下面的代码中演示了一些较复杂的绑定：

```
Rectangle {
  function calculateMyHeight() {
    return Math.max(otherItem.height, thirdItem.height);
  }

  anchors.centerIn: parent
  width: Math.min(otherItem.width, 10)
  height: calculateMyHeight()
  color: { if (width > 10) "blue"; else "red" }
}
```

从语法上来说，绑定是可以无限复杂的。但是，如果一个绑定变得非常复杂，例如涉及多行或者使用了循环，那么最好是可以重新来编写这个组件，或者是将绑定的内容写在一个单独的函数中。

### 1. 更改绑定

PropertyChanges 元素可以用在一个状态改变中来修改属性的绑定。下面的代码中，当在 square 状态时，修改了 Rectangle 的 width 属性的绑定为 otherItem.height。当返回到默认的状态时，width 属性便会恢复到以前的绑定：

```
Rectangle {
    id: rectangle
    width: otherItem.width
    height: otherItem.height

    states: State {
        name: "square"
        PropertyChanges {
            target: rectangle
            width: otherItem.height
        }
    }
}
```

### 2. 使用 JavaScript 进行属性分配的影响

使用 JavaScript 代码来为一个属性分配值，不会产生属性绑定，例如：

```
Rectangle {
    Component.onCompleted: {
        width = otherItem.width;
    }
}
```

这样只会简单地给 Rectangle 的 width 属性赋值，而不会创建属性绑定。而且还要注意，如果给已经进行了绑定的属性进行赋值，那么将会移除绑定。一个属性在一个时间只能拥有一个值，如果任何代码明确的为其设置了值，那么绑定将会被移除。下面代码中的 Rectangle 的 width 属性会为 13，而忽略了 otherItem 的 width：

```
Rectangle {
    width: otherItem.width

    Component.onCompleted: {
        width = 13;
    }
}
```

### 3. 绑定元素

前面介绍的隐式绑定语法是易于使用的，而且大多数的绑定都可以完美的工作。在一些高级的情况下，必须明确地使用绑定元素来创建一个绑定。例如，要将一个从

C++中暴露的属性(system.brightness)绑定到一个来自QML的值(slider.value),就需要使用到绑定元素:

```
Binding {
  target: system
  property: "brightness"
  value: slider.value
}
```

### 6.2.3 QML文件和QML组件

#### 1. QML文件

一个QML文件就是一块QML源代码。QML文件一般对应存储在硬盘上或者网络上的文件,不过也可以直接从文本数据进行构建。例如,上一节中的helloworld.qml文件就是一个QML文件。QML文件一般使用UTF-8格式来进行编码。一个QML文件总是以一个或者多个import语句开始。为了避免在以后版本导入的元素影响现在的QML文件,在一个文件中可用的元素类型使用导入的QML模块进行控制。也就是说,QML是一个版本语言。从语法上来说,一个QML文件是自包含的,QML不包含在将文件提交给QML运行环境之前对其进行修改的预处理器。import语句并不会包含任何代码到文件中,只是提示QML运行环境怎样来解析在文件中发现的类型引用。在QML文件中的任何类型引用,例如Rectangle和Text,都是完全基于import语句进行解析的。QML默认不会导入任何的模块,所以至少应该提供一个import语句,不然没有任何的元素可以被使用。在一个QML文件中的每一个id值都必须是唯一的,但是在不同的文件中可以有相同的id值,因为id值是在文件作用域进行解析的。

#### 2. QML文件作为组件定义

一个QML文件定义了一个独立的、顶级的QML组件(QML component)。一个QML组件就是一个模版,被QML运行环境解释来创建一个带有一些预定义行为的对象。因为它是一个模版,所以一个独立的QML组件可以运行多次来产生多个对象,每一个对象都可以称为该组件的实例。一旦创建,实例便不再依赖于创建它们的组件,所以它们可以在独立的数据上进行操作。下面创建一个Button组件(在Button.qml中定义),然后在application.qml中创建它的实例。

Button.qml文件的内容如下:

```
import QtQuick 1.0

Rectangle {
  property alias text: textItem.text
```

```
    width: 100; height: 62
    color: "blue"

    Text { id: textItem; color: "white" }
}
```

这里在最开始使用了属性别名 property alias 来为 textItem 对象的 text 属性重新指定了一个名称为 text，这样在组件外面才可以调用该属性，这个会在下一节详细讲解。下面是 application.qml 文件中的代码：

```
import QtQuick 1.0

Column {
    spacing: 10

    Button { text: "Apple" }
    Button { text: "Orange" }
    Button { text: "Pear" }
    Button { text: "Grape" }
}
```

这样便创建了 4 个 Button 组件的实例，它们分别拥有不同的文本，这是通过设置 text 属性完成的。因为 Button.qml 中使用了 Rectangle 元素作为根元素，所以在 Button 的实例中也可以使用 Rectangle 中所有的属性。可以看到，这里可以直接使用 Button 组件，而不需要包含任何的文件以及进行任何的设置，这是因为这里假定 Button.qml 与 helloworld.qml 放在同一个目录中，而在项目文件中已经将该目录设置为了默认的 QML 文件目录。

任何的 QML 代码片段都可以成为一个组件，只需要将它放入一个“<Name>.qml”文件中，这里的<Name>是一个新的元素名称，而且必须使用大写字母开头。对于同一目录中的其他 QML 组件和应用程序来说，新建的 QML 组件文件会自动成为新的 QML 元素类型。例如，这里可以像使用 Rectangle 元素一样来使用 Button 组件。

**3. 内联组件(inline Component)**

除了所有的 QML 文件定义的顶级组件，以及在独立的文件中的可重用组件以外，文件中也可以包含内联组件。内联组件使用 Component 元素声明，它们包含了常规顶级组件的所有特性。组件是 QML 中最重要的基本组成块之一，而且经常被其他元素用作“工厂”。例如，ListView 元素使用 delegate 组件作为模版来实例化列表条目，每一个列表条目都是该组件的一个新的实例，它们包含了各自特定的数据集：

```
import QtQuick 1.0
```

```
Rectangle {
  width: 240; height: 320;

  resources: [
    Component {
      id: contactDelegate
      Text {
        text: modelData.firstName + " " + modelData.lastName
      }
    }
  ]

  ListView {
    anchors.fill: parent
    model: contactModel
    delegate: contactDelegate
  }
}
```

与其他 QML 元素一样，Component 元素是一个对象，它必须分配到一个属性，Component 对象也拥有一个对象 id。在这里的代码中，内联组件添加到了 Rectangle 的 resources 列表中，然后使用属性绑定来将 Component 分配到 ListView 的 delegate 属性。虽然使用属性绑定可以共享 Component 对象(例如，QML 文件拥有多个 ListView 使用相同的 delegate)。在这段代码中，因为只有一个 ListView 元素，所以 Component 可以直接分配给 ListView 的 delegate。QML 语言在直接分配一个组件给一个属性时可以自动插入 Component 标签，所以下面两段代码，与前面的代码功能是一样的。

代码片段一：

```
import QtQuick 1.0

Rectangle {
  width: 240; height: 320;

  ListView {
    anchors.fill: parent
    model: contactModel
    delegate: Component {
      Text {
        text: modelData.firstName + " " + modelData.lastName
      }
    }
  }
}
```

代码片段二：

```
import QtQuick 1.0

Rectangle {
  width: 240; height: 320;

  ListView {
    anchors.fill: parent
    model: contactModel
    delegate: Text {
      text: modelData.firstName + " " + modelData.lastName
    }
  }
}
```

### 6.2.4　在组件中添加属性

QML 中的一个关键的概念就是可以自定义 QML 组件来适用于自己的应用程序。标准的 QML 元素为创建一个 QML 应用程序提供了必要的组件，除了这些，还可以编写自定义的组件并进行重用。当编写一个 QML 应用程序时，无论它规模的大小，都应该将 QML 代码分离成一些较小的组件来执行特定的功能，而不是将所有的代码都写在一个 QML 文件中，因为这样难以管理，而且还会包含重复的代码。

前一节已经创建了一个组件，不过它只包含了最简单的功能。要想编写一个有用的组件，一般必须为其提供一些自定义的属性来存储特定的数据并且与其进行通信。这是通过向组件中添加如下的内容来实现的：

- 属性：可以在外部进行访问来修改一个对象（例如，Item 拥有 width 和 height 属性），也可以用于属性绑定；
- 函数：包含 JavaScript 代码的函数可以在内部或者外部被调用（例如，Animation 拥有 start()函数）；
- 信号：当一个事件发生时用来提示其他对象（例如，MouseArea 拥有一个 clicked 信号）。

对应本节与下一小节的内容，可以在 Qt 帮助中查看 Writing QML Components: Properties, Methods and Signals 关键字。

**1. 添加属性**

一个属性就是 QML 组件中的一个值，可以被其他对象读取和修改。例如，Rectangle 组件拥有 width、height 和 color 属性。特别是，属性可以用于属性绑定，这样属性的值可以使用其他属性的值来自动更新。使用下面的语法来定义一个新的属性：

```
[default] property <type> <name>[: defaultValue]
```

一个属性声明可以出现在QML组件的任何地方，不过一般写在最顶部。一个组件中不可以出现相同名称的属性。（一个属性可以与相同类型的已经存在的属性使用相同的名称，不过不建议这样做，因为这样已经存在的属性会被隐藏而无法访问。）

在下面的代码中，ImageViewer组件（ImageViewer.qml文件）定义了一个*string类型的属性currentImage*，它的初始化值是“default－image.png”，这个属性被用在Image子对象中来设置一个图片：

```
import QtQuick 1.0

Item {
  id: item

  property string currentImage: "default - image.png"

  width: 200; height: 200

  Image { source: item.currentImage }
}
```

然后可以在其他文件比如application.qml中来创建一个ImageViewer对象，然后读取或者修改currentImage的值：

```
import QtQuick 1.0

ImageViewer {
  id: viewer

  currentImage: "http://qt.nokia.com/logo.png"

  Text { text: viewer.currentImage }
}
```

属性是否拥有默认的初始值是可选的。

**2. 支持的属性类型**

所有的QML属性都是有类型的。注意，属性的类型必须被声明，如在前面的代码中currentImage属性使用了string类型。类型用来决定属性的行为以及这个属性怎样使用C++来定义。默认支持多种属性类型，如表6－2所列，其中还列出了类型的默认值以及对应的C++类型。

表 6-2 QML 的属性类型

| QML 类型 | 默认值 | C++类型 |
|---|---|---|
| int | 0 | int |
| bool | false | bool |
| double | 0.0 | double |
| real | 0.0 | double |
| string | ""(空字符串) | QString |
| url | ""(空 url) | QUrl |
| color | #000000(黑色) | QColor |
| date | 未定义 | QDateTime |
| variant | 未定义 | QVariant |

### 3. 默认属性

声明属性时可选的 default 可以使该属性成为一个类型的默认属性。这允许其他对象来指定默认属性值来作为子元素。例如,Item 元素的默认属性是它的 children 属性,所以 Item 的孩子可以这样来设置:

```
Item {
  Rectangle {}
  Rectangle {}
}
```

如果 children 属性不是 Item 的默认属性,那么代码就必须这样写:

```
Item {
  children: [
    Rectangle {}
    Rectangle {}
  ]
}
```

指定默认的属性会覆盖已经存在的默认属性,在同一个类型中只能使用一次 default。

### 4. 属性别名

属性别名是更高级的属性声明形式。与定义一个属性不同,定义一个属性会为该属性分配新的、独立的存储空间,而属性别名会将新声明的属性(称为别名属性)作为一个已经存在的属性(被别名的属性)的直接引用。对别名属性的读/写操作等价于对被别名的属性进行读/写操作。属性别名的声明语法如下:

```
[default] property alias <name>: <alias reference>
```

因为别名属性与被别名的属性拥有相同的类型，所以这里省略了该类型，但是必须使用“alias”关键字。一个属性别名包含了一个强制的别名引用(alias reference)。这个别名引用用来定位被别名的属性。与属性绑定相似，别名引用的语法是高度受限的。别名引用可以使用下面两种格式中的一种：

```
<id>.<property>
<id>
```

这里的<id>必须是在同一个组件中的对象的id，对于可选的<property>需要是这个对象的一个属性，也就是说可以对一个对象或者这个对象的一个属性使用别名。例如在6.2.3小节的Buttom.qml中使用了text别名属性，它关联到了textItem对象的text属性。在外部修改Button组件的text属性，就会直接修改textItem.text的值。在这种情况下使用别名属性是很必要的，如果Button的text属性不是一个别名，那么改变它的值是无法自动改变textItem.text值的。这是因为属性绑定不是双向的：改变textItem.text的值会改变Button的text属性值，但是反过来却不可以。

属性别名对于允许外部对象直接修改和访问一个组件中的子对象是很有用的。例如，在下面的代码中将currentImage属性作为Image子对象的别名：

```
// ImageViewer.qml
import QtQuick 1.0

Item {
  id: item

  property alias currentImage: image

  width: 200; height: 200

  Image { id: image }
}
```

这样就可以在ImageViewer的对象中使用currentImage来直接访问和修改ImageViewer组件中的Image子对象和它的属性了：

```
// application.qml
import QtQuick 1.0

ImageViewer {
  id: viewer

  currentImage.source: "http://qt.nokia.com/logo.png"
  currentImage.width: width
```

```
    currentImage.height: height
    currentImage.fillMode: Image.Tile

    Text { text: currentImage.source }
}
```

很明显，使用这种方式来暴露子对象应该十分小心，因为这样允许外部对象随意的修改它们。但是，在一些特殊的情况下这种使用别名属性的方法是非常有用的。可以参考一下 Qt 自带的 Tab Widget Example 示例程序。使用属性别名要注意以下几点：

① 只有在指定它们的组件创建完成时别名才可用，这主要表现在组件本身在创建时不能直接使用别名属性。例如下面的代码是无法工作的：

```
property alias buttonText: textItem.text
buttonText: "Some text" // 当设置这个值的时候 buttonText 还没有被定义
```

② 别名引用不能使用在同一个组件中声明的另一个别名属性。例如下面的代码是无法工作的：

```
id: root
property alias buttonText: textItem.text
property alias buttonText2: root.buttonText
```

当组件被创建的时候，buttonText 的值还没有被分配，所以 root. buttonText 将会是一个未定义的值。

③ 一个别名属性可以和现有的属性使用相同的名称，例如，下面的代码中 color 别名属性和内建的 Rectangle::color 属性的名称相同：

```
Rectangle {
    property alias color: childRect.color
    color: "red"
    Rectangle { id: childRect }
}
```

在使用该组件的任何对象操作它的 color 属性时都是使用的别名，而不是一般的 Rectangle::color 属性。而在组件内部，Rectangle 可以正确使用真实定义的属性，并设置属性为 red，而不会使用别名。

### 6.2.5 在组件中添加函数和信号

#### 1. 添加函数

QML 组件中可以定义 JavaScript 代码的函数，这些函数可以在内部调用，也可以被其他对象调用。定义一个函数的语法如下：

```
function <name>([<parameter name>[, ...]]) { <body> }
```

这个声明可以出现在任何地方，不过一般是写在顶部。不可以在相同的类型块(比如Rectangle)中声明两个相同名称的函数。与信号不同，函数的参数类型不需要声明，它们默认是variant类型。函数体使用JavaScript进行编写，在其中可以访问参数。下面代码中的say()函数有一个text参数：

```
Rectangle {
  id: rect

  width: 100; height: 100

  function say(text) {
    console.log("You said: " + text);
  }

  MouseArea {
    anchors.fill: parent
    onClicked: rect.say("Mouse clicked")
  }
}
```

一个函数可以和一个信号进行关联，这样每当信号发射时都会自动调用这个函数。

**2. 添加信号**

当一个事件发生时，可以使用信号通知其他对象。例如，MouseArea的clicked信号通知其他对象在这个区域中已经单击过了。定义一个新的信号的语法如下：

```
signal <name>[([<type> <parameter name>[, ...]])]
```

这个声明可以出现在任何地方，不过一般是包含在顶部。同样，在一个类型块中是不能声明两个相同名称的信号的。下面有3个声明信号的例子：

```
Item {
  signal clicked
  signal hovered()
  signal performAction(string action, variant actionArgument)
}
```

如果信号没有参数，那么“()”括号便是可选的。如果使用了参数，那么就必须声明参数的类型。这里允许使用的参数类型如表6-2所列。向一个组件中添加一个信号，便会自动添加一个信号处理器，它使用“on<SignalName>”来命名，这里需要将信号名的第一个字母大写。比如代码中定义的3个信号对应的信号处理器分别为：

```
onClicked
onHovered
onPerformAction
```

发射信号就是简单调用它，就像操作函数一样。在下面的代码中，当单击 MouseArea 时，它会通过调用 rect. buttonClicked()来发射父对象的 buttonClicked 信号。这个信号会被 application. qml 通过 onButtonClicked 信号处理器获取：

```
// Button.qml
import QtQuick 1.0

Rectangle {
  id: rect

  signal buttonClicked

  width: 100; height: 100

  MouseArea {
    anchors.fill: parent
    onClicked: rect.buttonClicked()
  }
}
```

下面是 application. qml 文件的定义：

```
// application.qml
import QtQuick 1.0

Button {
  width: 100; height: 100
  onButtonClicked: console.log("Mouse was clicked")
}
```

如果信号包含参数，它们可以在信号处理器中使用。下面的代码中，buttonClicked 在发射时使用了 xPos 和 yPos 两个参数：

```
// Button.qml

Rectangle {
  id: rect

  signal buttonClicked(int xPos, int yPos)

  width: 100; height: 100
```

```
    MouseArea {
        anchors.fill: parent
        onClicked: rect.buttonClicked(mouse.x, mouse.y)
    }
}
```

下面是对应的 application.qml 文件的定义：

```
// application.qml

Button {
    width: 100; height: 100
    onButtonClicked: {
        console.log("Mouse clicked at " + xPos + "," + yPos)
    }
}
```

**3. 将信号关联到其他函数和信号上**

信号对象有一个 connect()函数，可以用来将一个信号关联到一个函数或者其他的信号上。当一个信号关联到一个函数上，这个函数会在信号发射时自动调用。(在Qt术语中，关联到一个信号上的函数被称为槽；在QML中定义的所有函数都被视为Qt槽。)这样信号会被这个函数接收而不是被信号处理器来接收。例如，前面的application.qml可以更改为这样：

```
Item {
    id: item

    width: 200; height: 200

    function myMethod() {
        console.log("Button was clicked!")
    }

    Button {
        id: button
        anchors.fill: parent
        Component.onCompleted: buttonClicked.connect(item.myMethod)
    }
}
```

这样无论何时信号被发射，都会调用 myMothod 函数。在大多数情况下，使用信号处理器来接收信号就足够了，因为这里的代码使用 connect()函数比以前使用onButtonClicked处理器没有任何的优势。然而，如果要动态创建对象或者集成JavaScript代码，就会发现 connect()函数很有用。例如，下面的组件中动态创建了3

个 Button，并将每一个对象的 buttonClicked 信号都关联到了 myMethod()函数上：

```
Item {
  id: item

  width: 300; height: 100

  function myMethod() {
    console.log("Button was clicked!")
  }

  Row { id: row }

  Component.onCompleted: {
    var component = Qt.createComponent("Button.qml")
    for (var i = 0; i<3; i++) {
      var button = component.createObject(row)
      button.border.width = 1
      button.buttonClicked.connect(myMethod)
    }
  }
}
```

使用同样的方法可以关联一个信号到一个动态创建的对象的函数上，或者关联一个信号到一个 JavaScript 函数上。

还有一个相应的 disconnect()函数用来移除关联的信号。下面的代码移除了在 application.qml 中创建的关联：

```
// application.qml
Item {
  ...
  function removeSignal() {
    button.clicked.disconnect(item.myMethod)
  }
}
```

另外，connect()函数也可以关联一个信号到另一个信号上，这样会产生“转发”信号效果：当相关信号被发射后它会自动被发射。例如，在前面 Button.qml 中 MouseArea 的 onClicked 处理器可以通过调用一个 connect()来代替：

```
MouseArea {
  anchors.fill: parent
  Component.onCompleted: clicked.connect(item.buttonClicked)
}
```

无论何时 MouseArea 的 clicked 信号被发射，rect.buttonClicked 信号也会自动

被发射。

### 6.2.6 集成JavaScript

QML推荐使用属性绑定和现有的QML元素来创建界面。为了允许执行更高级的行为，QML紧密集成了必要的JavaScript代码。QML中提供的JavaScript环境比在网页浏览器中的更严格。在QML中不可以添加或者修改JavaScript全局对象的成员，因为这样做可能会使用一个没有经过声明的变量。这在QML中会抛出一个异常，所以所有的局部变量都应该明确地声明。除了标准的JavaScript属性，在QML全局对象(在后面的6.8.4小节讲到)中还包含了一些很有用的函数，可以用来简化创建界面以及和QML环境进行交互。对应本节的内容，可以在帮助中查看Integrating JavaScript关键字。

**1. 内联JavaScript**

较小的JavaScript函数可以和其他QML声明一起写在QML组件中。这些内联函数会像一般的函数一样添加到QML元素中：

```
Item {
  function factorial(a) {
    a = parseInt(a);
    if (a <= 0)
      return 1;
    else
      return a * factorial(a - 1);
  }

  MouseArea {
    anchors.fill: parent
    onClicked: console.log(factorial(10))
  }
}
```

像一般的函数一样，在一个QML组件根元素中的内联函数可以在组件外被调用。如果不想被外部调用，那么这个函数可以添加到一个根元素以外的其他元素中，或者编写进一个外部的JavaScript文件中。

**2. 分离的JavaScript文件**

大块的JavaScript代码需要写在一个独立的文件中，这些文件可以通过使用import语句导入QML文件中，就像导入QML模块一样。例如，前面代码中的factorial()函数可以移动到一个外部名为“factorial.js”的文件中，然后像这样进行访问：

```
import "factorial.js" as MathFunctions
```

```
Item {
    MouseArea {
        anchors.fill: parent
        onClicked: console.log(MathFunctions.factorial(10))
    }
}
```

相对和绝对的 JavaScript 路径都可以被导入。如果脚本文件不可访问，那么将发生错误。如果 JavaScript 需要从一个网络资源中获取，那么组件的状态会被设置为 Loading，直到脚本被下载完毕。被导入的 JavaScript 文件总是使用 as 关键字来进行限定，每一个 JavaScript 文件的限定符必须是唯一的，在限定符和 JavaScript 文件之间是一对一映射的。

**3. 代码隐藏(Code－Behind)实施文件**

大多数的 JavaScript 文件被导入一个 QML 文件是有状态的，它们经常作为该 QML 文件的逻辑实现。在这种情况下，为了使 QML 组件的实例有正确的行为，每一个实例都需要有 JavaScript 对象和状态的一个独立的备份。导入一个 JavaScript 文件时的默认行为是为每一个 QML 组件实例提供一个唯一的、独立的备份。JavaScript 代码和 QML 组件实例运行在相同的范围，因此可以访问和操作对象和声明的属性。

**4. 无状态的 JavaScript 库**

一些 JavaScript 文件的行为更像库文件，它们提供了一组无状态的辅助函数来提供输入和计算输出，但是从来不直接操作 QML 组件实例。如果每一个 QML 组件实例都有一个这些库的拷贝，那么会造成浪费。JavaScript 程序员可以使用一个 pragma 来指明一个特定的文件是一个没有状态的库，例如：

```
// factorial.js
.pragma library

function factorial(a) {
    a = parseInt(a);
    if (a <= 0)
        return 1;
    else
        return a * factorial(a - 1);
}
```

这个 pragma 声明必须出现在除了注释以外的所有 JavaScript 代码以前。虽然 QML 值可以作为函数的参数进行传递，不过这些共享的、无状态的库文件不能够直接访问 QML 组件实例对象或者属性。

**5. 从其他 JavaScript 文件进行导入**

如果一个 JavaScript 文件需要使用定义在其他 JavaScript 文件中的函数，可以

通过使用Qt.include()函数来导入其他的文件。这样会将其他文件中的所有函数导入到当前文件的命名空间中。例如，下面的代码中调用了script.js文件中的showCalculations()，而在script.js中又调用了factorial.js中的factorial()函数，这是通过使用Qt.include()来实现包含factorial.js文件的：

```
import QtQuick 1.0
import "script.js" as MyScript

Item {
  width: 100; height: 100

  MouseArea {
    anchors.fill: parent
    onClicked: {
      MyScript.showCalculations(10)
      console.log("Call factorial() from QML:",
        MyScript.factorial(10))
    }
  }
}
```

下面是script.js文件的内容：

```
// script.js
Qt.include("factorial.js")

function showCalculations(value) {
  console.log("Call factorial() from script.js:",
    factorial(value));
}
```

下面是factorial.js文件的内容：

```
// factorial.js
function factorial(a) {
  a = parseInt(a);
  if (a <= 0)
    return 1;
  else
    return a * factorial(a - 1);
}
```

注意，调用Qt.include()将会从factorial.js导入所有的函数到MyScript命名空间，这也就意味着QML组件可以直接使用MyScript.factorial()来访问factorial()函数。

**6. 在启动时运行 JavaScript**

有时需要在应用程序(或者组件实例)启动时运行一些命令代码,但如果仅仅包含启动脚本作为全局代码,因为 QML 环境还没有完全建立起来,所以这样会有严重的局限,例如一些对象可能还没有被创建或者一些属性绑定还没有被运行。在后面讲述的 QML JavaScript 限制一段中涵盖了全局脚本代码的确切限制。QML 的 Component 元素提供了一个附加的 onCompleted 属性可以用来在 QML 环境完全建立以后切换到启动脚本代码的执行。例如:

```
Rectangle {
  function startupFunction() {
    // ... startup code
  }

  Component.onCompleted: startupFunction();
}
```

任何在 QML 文件中的元素,包含嵌套的元素和嵌套的 QML 组件实例,都可以使用这个附加属性。如果有多个 onCompleted()处理器在启动时执行,它们会以未定义的顺序依次执行。类似的,Component::onDestruction 附加属性会在组件销毁时触发。

**7. 属性赋值与属性绑定**

当同时使用 QML 和 JavaScript 时,区分 QML 属性绑定和 JavaScript 赋值是很重要的。在 QML 中,使用"属性:值"语法来创建一个属性绑定:

```
Rectangle {
  width: otherItem.width
}
```

每当 otherItem.width 更改时,Rectangle 的 width 也会自动更新。但是,例如下面的代码片段,它会在 Rectangle 被创建时执行:

```
Rectangle {

  Component.onCompleted: {
    width = otherItem.width;
  }
}
```

这为 Rectangle 的 width 分配了 otherItem.width 的值,它通过使用 JavaScript 中的"属性 = 值"语法实现的。与 QML 中的"属性:值"语法不同,这个不会调用 QML 的属性绑定;当执行代码时会为 Rectangle 的 width 属性分配 otherItem.width 的值,而当该值改变时不会自动更新。

### 8. 在JavaScript中接收QML信号

要接收一个QML信号，可以使用信号的connect()函数将它关联到一个JavaScript函数上。例如，下面的代码中将MouseArea的clicked信号关联到了script.js中的jsFunction()上：

```
import QtQuick 1.0
import "script.js" as MyScript

Item {
  id: item
  width: 200; height: 200

  MouseArea {
    id: mouseArea
    anchors.fill: parent
  }

  Component.onCompleted: {
    mouseArea.clicked.connect(MyScript.jsFunction)
  }
}
```

下面是script.js文件的内容：

```
// script.js

function jsFunction() {
  console.log("Called JavaScript function!")
}
```

现在，每当MouseArea的clicked信号被发射，都会调用jsFunction()函数。

### 9. QML JavaScript限制

QML执行标准的JavaScript代码，会有下面的限制：

(1)JavaScript代码不能修改全局对象

在QML中，全局对象是一个常量，现有的属性不能够被修改和删除，也不能够创建新的属性。大多数的JavaScript程序并不是有意修改全局对象的，然而，JavaScript自动生成一个未声明的变量是全局对象的隐式修改，这在QML中是禁止的。例如，假设变量a在作用域链中不存在，那么下面的代码在QML中是非法的：

```
// 非法修改未声明的变量
a = 1;
for (var ii = 1; ii < 10; ++ii)
  a = a * ii;
```

```
console.log("Result: " + a);
```

它可以这样简单修改为合法的代码：

```
var a = 1;
for (var ii = 1; ii < 10; ++ii)
  a = a * ii;
console.log("Result: " + a);
```

无论隐式的或者显式的对全局对象的修改都会导致一个异常。

(2)全局代码运行在一个缩小的范围

在启动时，如果 QML 文件包含一个外部的 JavaScript 文件和“全局”代码，它会在只包含该外部文件和这个全局对象的范围内执行。也就是说，它不会再像通常那样访问 QML 对象和属性。全局代码只访问脚本中的局部变量是允许的，下面是一个有效的全局代码：

```
var colors = [ "red", "blue", "green", "orange", "purple" ];
```

全局代码访问 QML 对象将无法正常运行：

```
// 非法的全局代码 - "rootObject" 变量未定义
  var initialPosition = { rootObject.x, rootObject.y }
```

存在此限制是因为 QML 环境尚未被完全建立。要在环境完全启动后运行代码，可以参考第 6 部分的内容。

(3)目前在 QML 中 this 值是未定义的

在 QML 中 this 值是未定义的，要引用任何元素，可以使用其 id。例如：

```
Item {
  width: 200; height: 100
  function mouseAreaClicked(area) {
    console.log("Clicked in area at: " + area.x + ", " + area.y);
  }

  // 因为 this 未定义，所以下面的代码不会工作
  MouseArea {
    height: 50; width: 200
    onClicked: mouseAreaClicked(this)
  }

  // 这样可以将 area2 传递给函数
  MouseArea {
    id: area2
    y: 50; height: 50; width: 200
    onClicked: mouseAreaClicked(area2)
```

```
    }
}
```

### 6.2.7 QML动态对象管理

QML提供了很多方法来动态创建和管理QML对象。如Loader、Repeater、ListView、GridView和PathView等元素都支持动态对象管理。对象也可以在C++中被创建和管理，这是混合QML/C++应用程序的首选方式，这个可以参考6.8节的内容。QML也支持在JavaScript代码中动态创建对象，这在现有的QML元素不适合应用程序需要的情况下是很有用的，而且也不需要涉及C++组件。对应本节内容，可以在Qt帮助中查看Dynamic Object Management in QML关键字，还可以参考Qt提供的Dynamic Scene example演示程序。也可以使用Loader来动态加载组件，Loader元素在6.5.4小节讲到。

**1. 动态创建对象**

这里有两种方法从JavaScript动态创建对象。既可以调用Qt.createComponent()来动态创建Compont对象，也可以使用Qt.createQmlObject()从QML字符串来创建对象。如果已经有一个在.qml文件中定义的组件，而且希望动态创建该组件的一个实例，那么使用第一种方法是比较好的；如果QML本身是在运行时产生的，那么使用QML字符串来创建对象是很有用的。

(1)动态创建一个组件

要动态加载定义在一个QML文件中的组件，可以在QML全局对象上调用Qt.createComponent()函数。这个函数需要将QML文件的URL作为其参数，然后从这个URL上创建一个Component对象。一旦有了一个Component，就可以调用它的createObject()函数来创建该组件的一个实例。这个函数需要指定新对象的父对象。因为图形项目没有父对象是无法显示在场景上的，所以建议这样来设置父对象。然而，如果想稍后再为其设置父对象，可以安全的设置null作为该函数的参数。

下面来看一个例子。首先是Sprite.qml文件，它定义了一个简单的QML组件：

```
import QtQuick 1.0

Rectangle { width: 80; height: 50; color: "red" }
```

下面是主应用程序文件main.qml，导入了JavaScript文件componentCreation.js来创建Sprite对象：

```
import QtQuick 1.0
import "componentCreation.js" as MyScript

Rectangle {
```

```
    id: appWindow
    width: 300; height: 300

    Component.onCompleted: MyScript.createSpriteObjects();
}
```

下面是 componentCreation.js 文件，其中在调用 createObject()以前检查了组件的状态是否为 Component.Ready，因为如果 QML 文件是从网络上加载的，那么它不会立即可用：

```
var component;
var sprite;

function createSpriteObjects() {
    component = Qt.createComponent("Sprite.qml");
    if (component.status == Component.Ready)
        finishCreation();
    else
        component.statusChanged.connect(finishCreation);
}

function finishCreation() {
    if (component.status == Component.Ready) {
        sprite = component.createObject(appWindow);
        if (sprite == null) {
            // 错误处理
        } else {
            sprite.x = 100;
            sprite.y = 100;
            // ...
        }
    } else if (component.status == Component.Error) {
        // 错误处理
        console.log("Error loading component:", component.errorString());
    }
}
```

如果可以确保 QML 文件是从本地文件加载的，那么可以忽略 finishCreation()函数，而在 createSpriteObjects()函数中立即调用 createObject()函数，例如上面的代码可以写成：

```
function createSpriteObjects() {
    component = Qt.createComponent("Sprite.qml");
    sprite = component.createObject(appWindow);
```

```
    }
}
```

### 6.2.7 QML动态对象管理

QML提供了很多方法来动态创建和管理QML对象。如Loader、Repeater、ListView、GridView和PathView等元素都支持动态对象管理。对象也可以在C++中被创建和管理，这是混合QML/C++应用程序的首选方式，这个可以参考6.8节的内容。QML也支持在JavaScript代码中动态创建对象，这在现有的QML元素不适合应用程序需要的情况下是很有用的，而且也不需要涉及C++组件。对应本节内容，可以在Qt帮助中查看Dynamic Object Management in QML关键字，还可以参考Qt提供的Dynamic Scene example演示程序。也可以使用Loader来动态加载组件，Loader元素在6.5.4小节讲到。

**1. 动态创建对象**

这里有两种方法从JavaScript动态创建对象。既可以调用Qt.createComponent()来动态创建Compont对象，也可以使用Qt.createQmlObject()从QML字符串来创建对象。如果已经有一个在.qml文件中定义的组件，而且希望动态创建该组件的一个实例，那么使用第一种方法是比较好的；如果QML本身是在运行时产生的，那么使用QML字符串来创建对象是很有用的。

(1)动态创建一个组件

要动态加载定义在一个QML文件中的组件，可以在QML全局对象上调用Qt.createComponent()函数。这个函数需要将QML文件的URL作为其参数，然后从这个URL上创建一个Component对象。一旦有了一个Component，就可以调用它的createObject()函数来创建该组件的一个实例。这个函数需要指定新对象的父对象。因为图形项目没有父对象是无法显示在场景上的，所以建议这样来设置父对象。然而，如果想稍后再为其设置父对象，可以安全的设置null作为该函数的参数。

下面来看一个例子。首先是Sprite.qml文件，它定义了一个简单的QML组件：

```
import QtQuick 1.0

Rectangle { width: 80; height: 50; color: "red" }
```

下面是主应用程序文件main.qml，导入了JavaScript文件componentCreation.js来创建Sprite对象：

```
import QtQuick 1.0
import "componentCreation.js" as MyScript

Rectangle {
```

```
    id: appWindow
    width: 300; height: 300

    Component.onCompleted: MyScript.createSpriteObjects();
}
```

下面是 componentCreation.js 文件，其中在调用 createObject()以前检查了组件的状态是否为 Component.Ready，因为如果 QML 文件是从网络上加载的，那么它不会立即可用：

```
var component;
var sprite;

function createSpriteObjects() {
    component = Qt.createComponent("Sprite.qml");
    if (component.status == Component.Ready)
        finishCreation();
    else
        component.statusChanged.connect(finishCreation);
}

function finishCreation() {
    if (component.status == Component.Ready) {
        sprite = component.createObject(appWindow);
        if (sprite == null) {
            // 错误处理
        } else {
            sprite.x = 100;
            sprite.y = 100;
            // ...
        }
    } else if (component.status == Component.Error) {
        // 错误处理
        console.log("Error loading component:", component.errorString());
    }
}
```

如果可以确保 QML 文件是从本地文件加载的，那么可以忽略 finishCreation()函数，而在 createSpriteObjects()函数中立即调用 createObject()函数，例如上面的代码可以写成：

```
function createSpriteObjects() {
    component = Qt.createComponent("Sprite.qml");
    sprite = component.createObject(appWindow);
```

```
    if (sprite == null) {
      // Error Handling
  console.log("Error creating object");
    } else {
      sprite.x = 100;
      sprite.y = 100;
      // ...
    }
}
```

注意，这里createObject()使用了appWindow作为参数，所以创建的对象会成为main.qml中appWindow的子对象。当使用相对路径来加载文件时，需要是相对于执行Qt.createComponent()的文件的路径。将信号关联到动态创建的对象上，或者从动态创建的对象上接收信号，都要使用信号的connect()函数。这个可以参考第6.2.5小节的第3部分内容。

(2)从QML字符串创建一个对象

如果QML直到运行时才被定义，可以使用Qt.createQmlObject()函数从一个QML字符串创建一个QML对象。例如：

```
var newObject = Qt.createQmlObject(
    'import QtQuick 1.0; Rectangle {color: "red"; width: 20; height: 20}',
    parentItem, "dynamicSnippet1");
```

第一个参数是要创建的QML字符串，就像一个新的文件一样，需要导入所需要的类型；第二个参数是父对象，需要是在场景中已经存在的一个对象；第三个参数是与新对象相关的文件的路径，用来报告错误。如果在QML字符串中导入的文件使用的是相对路径，那么需要相对于定义父对象的文件的路径。

**2. 维护动态创建的对象**

当管理动态创建的对象时，必须确保创建上下文(creation context)不会在创建的对象销毁前被销毁。否则，如果创建上下文被首先销毁，那么在动态创建对象中的绑定将不会再工作。实际的创建上下文依赖于对象是怎样被创建的：

- 如果使用了Qt.createComponent()，创建上下文就是调用该函数的QDeclarativeContext；
- 如果使用了Qt.createQmlObject()，创建上下文就是父对象的上下文；
- 如果定义了一个Component{}，然后在其上调用了createObject()，创建上下文就是该Component中定义的上下文。

另外需要注意，虽然动态创建的对象可以像其他对象一样来使用，但是它们没有id值。

**3. 动态删除对象**

在很多用户界面中，将图形项的透明度设置为0或者将其移出屏幕就够了，而不

需要将其删除。然而，如果有很多动态生成的对象，那么将不用的对象删除会得到一个很大的性能提升。不过应该注意，永远不要手动删除通过 QML 元素（例如 Loader 和 Repeater）动态生成的对象，不要删除不是自己动态创建的对象。

可以使用 destroy()函数来删除对象，这个函数有一个可选的参数，可以用来设置在销毁该对象以前的以毫秒为单位的延迟时间，该参数默认为 0。在下面的例子中，application. qml 中创建了 SelfDestroyingRect. qml 组件的 5 个实例，每一个实例运行一个 NumberAnimation，当动画结束时在其根对象上调用 destroy()来进行自我销毁。

下面是 application. qml 文件：

```
import QtQuick 1.0

Item {
  id: container
  width: 500; height: 100

  Component.onCompleted: {
    var component = Qt.createComponent("SelfDestroyingRect.qml");
    for (var i = 0; i<5; i++) {
      var object = component.createObject(container);
      object.x = (object.width + 10) * i;
    }
  }
}
```

下面是 SelfDestroyingRect. qml 文件的内容：

```
import QtQuick 1.0

Rectangle {
  id: rect
  width: 80; height: 80
  color: "red"

  NumberAnimation on opacity {
    to: 0
    duration: 1000
    onRunningChanged: {
      if (! running) {
        console.log("Destroying...")
        rect.destroy();
      }
    }
  }
}
```

另外，application.qml可以调用object.destroy()来销毁创建的对象。注意，在一个对象内部调用destroy()来自我销毁是安全的。对象不会在destroy()被调用时就被销毁，而是在该脚本块的末尾和下一帧之间的某个时间进行清理(除非将延时指定了一个非零值)。

还应该注意，如果SelfDestroyingRect实例被静态的创建，例如：

```
Item {
  SelfDestroyingRect { ... }
}
```

这样将会产生错误。因为只有动态创建的对象才可以被动态删除。使用Qt.createQmlObject()创建的对象可以相似的使用destroy()来删除：

```
var newObject = Qt.createQmlObject(
    'import QtQuick 1.0; Rectangle {color: "red"; width: 20; height: 20}',
    parentItem, "dynamicSnippet1");
newObject.destroy(1000);
```

## 6.2.8 QML的作用域

QML属性绑定、内联函数和导入的JavaScript文件都运行在一个JavaScript作用域。作用域控制表达式可以访问哪些变量，以及当两个或多个名字冲突时，哪个变量优先。因为JavaScript的内建作用域机制非常简单，QML对其进行了加强，使其更加自然的适应QML语言的扩展。对应本节内容，可以在Qt帮助中查看QML Scope关键字。

### 1. JavaScript作用域

QML的作用域扩展并没有干扰JavaScript本身的作用域。JavaScript程序员可以在编写函数、属性绑定或者在QML中导入JavaScript文件时重复使用现有的知识。在下面的例子中，addConstant()函数将会在传递过去的参数上加13，正像程序员所期望的那样，结果与QML对象的a和b属性值无关：

```
QtObject {
  property int a: 3
  property int b: 9

  function addConstant(b) {
    var a = 13;
    return b + a;
  }
}
```

QML遵循JavaScript一般的作用域规则，甚至在应用绑定时也是这样。下面的

代码将会为 a 属性绑定一个值 12：

```
QtObject {
  property int a

  a: { var a = 12; a; }
}
```

每一个在 QML 中的 JavaScript 表达式、函数或者文件都有它们自己唯一的变量对象，在它们任意一个里面声明的局部变量都不会和在另外一个里面声明的局部变量冲突。

**2. 元素名称和导入的 JavaScript 文件**

QML 文件包含了导入语句来定义元素名称和 JavaScript 文件，使其在文件中可见。除了在 QML 声明时使用，在访问附加属性和枚举值时 JavaScript 代码也会使用元素名称。在 QML 中 import 会影响到每一个属性绑定、QML 文件中的 JavaScript 函数以及那些嵌套的内联组件。下面的代码中显示了一个简单的 QML 文件，其中访问了一些枚举值，而且调用了一个导入的 JavaScript 函数：

```
import QtQuick 1.0
import "code.js" as Code

ListView {
  snapMode: ListView.SnapToItem

  delegate: Component {
    Text {
      elide: Text.ElideMiddle
      text: "A really, really long string that will require eliding."
      color: Code.defaultColor()
    }
  }
}
```

**3. 绑定的作用域对象**

属性绑定是在 QML 中最常见的 JavaScript 应用，关联了一个 JavaScript 表达式的结果和对象的一个属性。绑定对象所属的对象被称为绑定的作用域对象。在下面的代码中 Item 对象就是一个绑定的作用域对象：

```
Item {
  anchors.left: parent.left
}
```

绑定可以不加限制地访问作用域对象的属性。在前面的例子中，绑定可以直接

访问 Item 的 parent 属性，而不需要任何形式的对象前缀。QML 为 JavaScript 引入了一个更加结构化、面向对象的方式，因此不再需要使用 JavaScript 的 this 属性。

当从绑定中访问附加属性时要非常小心，因为它们会与作用域对象交互。从概念上讲，附加属性在所有对象上都存在，即使它们只对这些对象的子集有影响。因此，非限定的附加属性的读取，总会得到作用域对象附加属性的值，这并不是程序员的意图。例如，PathView 元素会向它的委托附加一个插值属性，这个插值依赖于在路径中具体的位置。因为 PathView 只会向委托的根元素附加这些属性，任何的子元素要访问这些属性都要明确限定根元素，就像下面的代码所展示的：

```
PathView {
    delegate: Component {
        Rectangle {
            id: root
            Image {
                scale: root.PathView.scale
            }
        }
    }
}
```

如果 Image 元素忽略了 root 前缀，那么它就会在无意中访问在它自己上的未设置的 PathView.scale 附加属性。

**4. 组件作用域**

在 QML 文件中的每一个 QML 组件都定义了一个逻辑作用域。每一个文件都至少有一个根组件，但是也可以拥有其他的内联子组件。组件的作用域是组件内的对象 id 和组件的根元素的属性的联合，例如：

```
Item {
    property string title

    Text {
        id: titleElement
        text: "<b>" + title + "</b>"
        font.pixelSize: 22
        anchors.top: parent.top
    }

    Text {
        text: titleElement.text
        font.pixelSize: 18
        anchors.bottom: parent.bottom
    }
}
```

这里的组件中先在上面显示了一个富文本标题字符串，然后在下面显示了相同文本的副本。第一个 Text 元素在构造显示的文本时直接访问了组件的 title 属性，根元素的属性可以直接被访问使得在整个组件中都可以分配数据。第二个 Text 元素使用了一个 id 来直接访问第一个 Text 的文本。由于 ID 由 QML 程序员明确指定，所以它们总是优先于其他属性名称。例如，如果在这个例子中绑定的作用域对象也有一个 titleElement 属性，那么依然优先考虑 id 为 titleElement，而不会把它当作一个属性。

**5. 组件实例的层次**

在 QML 中，组件实例将它们的作用域关联在一起形成了一个作用域层次。组件实例可以直接访问其祖先的作用域。例如，使用内联子组件时，它的组件作用域隐式的设置为了其外围组件作用域的孩子：

```
Item {
  property color defaultColor: "blue"

  ListView {
    delegate: Component {
      Rectangle {
        color: defaultColor
      }
    }
  }
}
```

组件实例层次允许委托组件的实例访问 Item 元素的 defaultColor 属性。当然，如果委托组件也有一个名为 defaultColor 的属性，那么将会优先访问它。

组件实例作用域层次可以扩展到非内联的组件。例如，在下面的代码中，TitlePage.qml 组件创建了两个 TitleText 实例，即使 TitleText 元素是在一个独立的文件中，当它在 TitlePage 中使用时依然可以访问 title 属性。注意，QML 是一个动态作用域语言，依赖于在哪里被使用。

下面是 TitlePage.qml 文件的内容：

```
import QtQuick 1.0
Item {
  property string title

  TitleText {
    size: 22
    anchors.top: parent.top
  }

  TitleText {
```

```
    size: 18
    anchors.bottom: parent.bottom
  }
}
```

下面是 TitleText.qml 文件的内容：

```
import QtQuick 1.0
Text {
  property int size
  text: "<b>" + title + "</b>"
  font.pixelSize: size
}
```

动态作用域是非常强大的，但是必须谨慎使用以避免 QML 代码的行为变得难以预料。一般情况下，它只用于两个组件已经通过其他方法紧密耦合的情况下。当构建可重用组件时，最好使用属性接口，例如：

下面是 TitlePage.qml 文件的内容：

```
import QtQuick 1.0
Item {
  id: root
  property string title

  TitleText {
    title: root.title
    size: 22
    anchors.top: parent.top
  }

  TitleText {
    title: root.title
    size: 18
    anchors.bottom: parent.bottom
  }
}
```

下面是 TitleText.qml 文件的内容：

```
import QtQuick 1.0
Text {
  property string title
  property int size

  text: "<b>" + title + "</b>"
  font.pixelSize: size
```

```
}
```

**6. JavaScript 全局对象**

除了 JavaScript 全局对象的所有属性以外，QML 还添加了一些自定义的扩展来更容易完成 UI 或者 QML 指定的任务。这些扩展可以参考 6.8.4 小节的内容。QML 通过不允许元素、id 和属性名称与全局对象的属性同名来防止冲突，例如，程序员可以确信 Math.min(10，9)总是可以像所期望的那样工作。

## 6.2.9 QML 的国际化

在 QML 中可以使用 qsTr()、qsTranslate()、QT_TR_NOOP()和 QT_TRANSLATE_NOOP 等函数将字符串标记为可翻译的。例如：

```
Text { text: qsTr("Pictures") }
```

这些函数是标准的 QtScript 函数，在 QScriptEngine::installTranslatorFunctions()函数的帮助文档处可以查看详细信息。下面是一个简单的例子。

首先创建一个包含要翻译文本的 QML 文件，这里是 hello.qml 文件：

```
import QtQuick 1.0

Rectangle {
  width: 200; height: 200
  Text { text: qsTr("Hello"); anchors.centerIn: parent }
}
```

下面使用 lupdate 来创建一个翻译源文件：

```
lupdate hello.qml -ts hello.ts
```

然后在 Qt 语言家中打开 hello.ts 文件，完成翻译并创建 hello.qm 发布文件。最后，可以测试翻译：

```
qmlviewer -translation hello.qm hello.qml
```

对应本节的内容，可以在 Qt 帮助中查看 QML Internationalization 关键字。翻译的详细过程可以参考《Qt Creator 快速入门》中第 9 章的内容，还可以查看 Qt 自带的 QML Internationalization example 示例程序。

## 6.2.10 QML 的编码约定

QML 的参考文档和示例程序中使用了相同的编码约定，为了风格的统一和代码的规范，建议读者以后编写 QML 代码时也遵循这个约定。对应本节的内容，可以在 Qt 帮助中查看 QML Coding Conventions 关键字。

### 1. QML对象

QML对象一般使用下面的顺序进行构造：

- id；
- 属性声明；
- 信号声明；
- JavaScript函数；
- 对象属性；
- 子对象；
- 状态；
- 状态切换。

为了获取更好的可读性，建议在不同部分之间添加一个空行。下面使用一个Photo对象作为示例：

```
Rectangle {
id: photo                                    // id放在第一行

  property bool thumbnail: false             // 属性声明
  property alias image: photoImage.source

  signal clicked                             // 信号声明

  function doSomething(x) {                  // javascript函数
    return x + photoImage.width
  }

  x: 20; y: 20; width: 200; height: 150      // 对象属性
  color: "gray"                              // 相关属性放在一起

  Rectangle {                                // 子对象
    id: border
    anchors.centerIn: parent; color: "white"

    Image { id: photoImage; anchors.centerIn: parent }
  }

  states: State {                            // 状态
    name: "selected"
    PropertyChanges { target: border; color: "red" }
  }

  transitions: Transition {                  // 切换
    from: ""; to: "selected"
```

```
    ColorAnimation { target: border; duration: 200 }
  }
}
```

**2. 分组属性**

如果使用了一组属性中的多个属性，那么使用组表示法，而不要使用点表示法，这样可以提高可读性。例如：

```
Rectangle {
  anchors.left: parent.left; anchors.top: parent.top;
  anchors.right: parent.right; anchors.leftMargin: 20
}

Text {
  text: "hello"
  font.bold: true; font.italic: true; font.pixelSize: 20;
  font.capitalization: Font.AllUppercase
}
```

可以写成这样：

```
Rectangle {
  anchors { left: parent.left; top: parent.top; right: parent.right;
    leftMargin: 20 }
}
Text {
  text: "hello"
  font { bold: true; italic: true; pixelSize: 20;
    capitalization: Font.AllUppercase }
}
```

**3. 私有属性**

QML 和 JavaScript 中并没有 C++中那样强制私有属性。例如，当这些属性是实施的一部分时，这里需要隐藏那些私有属性。作为惯例，私有属性使用两个下划线开头。例如，__area 是一个可以被访问但是并不作为公有使用的属性：

```
Item {
  id: component
  width: 40; height: 50
  property real __area: width * height * 0.5      // 并不是作为外部使用
}
```

**4. 列　表**

如果一个列表只包含一个元素，那么通常忽略方括号。例如下面的代码：

```
states: [
  State {
    name: "open"
    PropertyChanges { target: container; width: 200 }
  }
]
```

可以写成：

```
states: State {
  name: "open"
  PropertyChanges { target: container; width: 200 }
}
```

**5. JavaScript 代码**

如果脚本是一个单独的表达式，建议将它写成内联的：

```
Rectangle { color: "blue"; width: parent.width / 3 }
```

如果脚本只有几行，那么建议写成一块：

```
Rectangle {
  color: "blue"
  width: {
    var w = parent.width / 3
    console.debug(w)
    return w
  }
}
```

如果脚本有很多行，或者需要被不同的对象使用，那么建议创建一个函数，然后像下面这样来调用它：

```
function calculateWidth(object)
{
  var w = object.width / 3
  // ...
  // more javascript code
  // ...
  console.debug(w)
  return w
}

Rectangle { color: "blue"; width: calculateWidth(parent) }
```

如果是很长的脚本，可以将这个函数放在独立的 JavaScript 文件中，然后像下面这样来导入它：

```
import "myscript.js" as Script

Rectangle { color: "blue"; width: Script.calculateWidth(parent) }
```

# 6.3　QML 中的布局管理

## 6.3.1　使用 QML 定位器和重复器

定位器项(在本章中元素、项、项目视为同义词)是一个容器项,可以用来管理在声明式用户接口中项的位置和大小。定位器与标准 Qt 部件中的布局管理器很相似。当需要使用常规的布局来管理众多的项时可以使用定位器和重复器。对应本节的内容,可以在 Qt 帮助中查看 Using QML Positioner and Repeater Items 关键字,也可以参考各个元素的帮助文档。关于定位器的使用,还可以参考一下 Qt 自带的 Positioners Example 示例程序。

**1. 定位器**

基本 Qt Quick 图形元素集里提供了一组标准的定位器,分别是 Column、Row、Grid 和 Flow。下面分别进行介绍。

(1)Column

Column 项用来垂直排列项目并且使它们不会重叠。下面的例子中使用了 Column 项定位了几个形状不同的 Rectangle。Column 元素的 spacing 属性用来在这几个 Rectangle 之间添加间距。(项目源码路径:src\6\6 - 2\myColumn)

```
import QtQuick 1.0

Column {
  spacing: 2

  Rectangle { color: "red"; width: 50; height: 50 }
  Rectangle { color: "green"; width: 20; height: 50 }
  Rectangle { color: "blue"; width: 50; height: 20 }
}
```

代码的运行效果如图 6 - 9 所示。

(2)Row

Row 项用来水平排列项目并且使它们不会重叠。下面的例子中使用了一个 Row 来布局几个不同形状的 Rectangle。(项目源码路径:src\6\6 - 3\myRow)

```
import QtQuick 1.0
```

```
Row {
    spacing: 2

    Rectangle { color: "red"; width: 50; height: 50 }
    Rectangle { color: "green"; width: 20; height: 50 }
    Rectangle { color: "blue"; width: 50; height: 20 }
}
```

代码的运行效果如图 6-10 所示。

(3)Grid

Grid 项用来定位它的子项目,使它们排列在一个网格中而不会重叠。网格定位器会计算一个足够大的矩形单元网格来容纳所有的项目。向单元中放置项目,会从左向右、从上向下进行排列。每一个项目都放置在单元的左上角(0, 0)的位置。一个 Grid 默认有 4 列,而可以有无限多的行来容纳所有的子项目。行数和列数也可以通过 rows 和 columns 属性来指定。也可以通过 spacing 属性来设置子项目之间的间距,在水平方向和垂直方向会使用相同的间距。如下例。(项目源码路径:src\6\6-4\myGrid)

```
import QtQuick 1.0

Grid {
    columns: 3
    spacing: 2

    Rectangle { color: "red"; width: 50; height: 50 }
    Rectangle { color: "green"; width: 20; height: 50 }
    Rectangle { color: "blue"; width: 50; height: 20 }
    Rectangle { color: "cyan"; width: 50; height: 50 }
    Rectangle { color: "magenta"; width: 10; height: 10 }
}
```

代码的运行效果如图 6-11 所示。

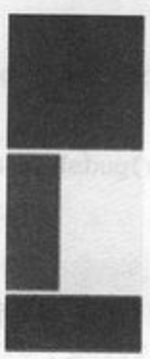


图 6-9 Column 运行效果

图 6-10 Row 运行效果

图 6-11 Grid 运行效果

(4)Flow

Flow 项用来将像单词一样的项目放置在一个页面上,通过换行使这些项目排列成多个行或列并且不会重叠。Flow 项排列项目的规则与 Grid 项是相似的。Flow 有一个 flow 属性,它是枚举类型,包含两个值,Flow. LeftToRight(默认)和 Flow. TopToBottom。前者是项目从左向右相邻排列,直到超出 Flow 的宽度,然后换到下一行;而后者是项目从上向下相邻排列,直到超出 Flow 的高度,然后换到下一列。在下面的例子中,显示了一个包含多个 Text 子项目的 Flow 项。(项目源码路径:src\6\6 - 5\myFlow)

```
import QtQuick 1.0

Rectangle {
  color: "lightblue"
  width: 300; height: 200

  Flow {
    anchors.fill: parent
    anchors.margins: 4
    spacing: 10

    Text { text: "Text"; font.pixelSize: 40 }
    Text { text: "items"; font.pixelSize: 40 }
    Text { text: "flowing"; font.pixelSize: 40 }
    Text { text: "inside"; font.pixelSize: 40 }
    Text { text: "a"; font.pixelSize: 40 }
    Text { text: "Flow"; font.pixelSize: 40 }
    Text { text: "item"; font.pixelSize: 40 }
  }
}
```

Grid 和 Flow 定位器的主要不同之处是在 Flow 中的项目当超出边界后会自动换行。例如更改 Flow 的大小,代码运行时可能出现如图 6 - 12 所示的效果。

**2. 重复器 Repeater**

Repeater 元素用来创建大量相似的项目。像其他视图元素一样,一个 Repeater 包含一个模型 model 和一个委托 delegate 属性:委托用来将模型中的每一个条目分别实例化。Repeater 通常会包含在定位器中以直观地对 Repeater 产生的众多委托项目进行布局。下面的例子中显示了一个重复器和一个 Grid 项目一起使用来排列一组 Rectangle 项目。(项目源码路径:src\6\6 - 6\myRepeater)

```
import QtQuick 1.0

Rectangle {
```

```
    width: 400; height: 400; color: "black"

    Grid {
      x: 5; y: 5
      rows: 5; columns: 5; spacing: 10

      Repeater { model: 24
        Rectangle { width: 70; height: 70
                    color: "lightgreen"

                    Text { text: index
                      font.pointSize: 30
                      anchors.centerIn: parent } }
      }
    }
}
```

Repeater中创建的项目数量可以通过count属性获得，该属性是只读的。代码的运行效果如图6-13所示。

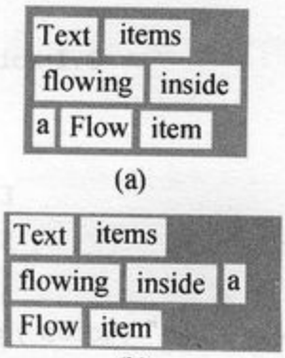



图6-12 Flow运行效果

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | |

图6-13 Repeater运行效果

### 3. 使用切换

切换可以使在定位器中进行添加、移入或者删除项目时具有动画效果。4个定位器元素都有一个add和move属性，都需要分配一个Transition对象。add切换应用在向定位器中创建一个项目以及将项目通过更换父对象而变为定位器的孩子时；move切换应用在删除定位器中的一个项目以及通过更换父对象而从定位器中移除对象时。此外，将项目的透明度更改为0时会使用move切换使项目隐藏；当使项目的透明度为非0时，会使用add切换使项目显示。定位器切换只会影响项目的位置(x, y)。下面的代码片段演示了在Flow中启用move切换：

```
Flow {
  id: positioner

  move: Transition {
```

```
    NumberAnimation {
      properties: "x,y"
      ease: "easeOutBounce"
    }
  }
}
```

对于其他定位器，也可以使用相似的方法来启用切换。对于 Transition 和 NumberAnimation 元素的使用会在 6.6.5 小节讲解。

## 6.3.2 QML 中基于锚的布局

除了前面讲解的传统的 Grid、Row 和 Column 等，QML 还提供了一种使用锚(anchor)的概念来进行项目布局的方法。每一个项目都可以认为有一组无形的“锚线”：left、horizontalCenter、right、top、verticalCenter、baseline 和 bottom，如图 6-14 所示。这些锚线分别对应了 Item 元素中的 anchors.left 等属性，因为 QML 中所有可见的项目都继承自 Item 元素，所以所有的可视项目都可以使用锚来布局。锚系统也允许为一个项目的锚指定边距(margin)和偏移(offset)。边距指定了项目锚到外边界的空间量，而偏移允许使用中心锚线来定位。一个项目可以通过 leftMargin、rightMargin、topMargin 和 bottomMargin 来独立的指定锚边距，如图 6-15 所示，也可以使用 anchor.margins 来为所有的 4 个边指定相同的边距。锚偏移可以使用 horizontalCenterOffset、verticalCenterOffset 和 baselineOffset 来指定。

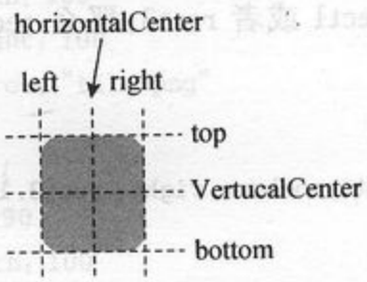



图 6-14　项目锚线示意图

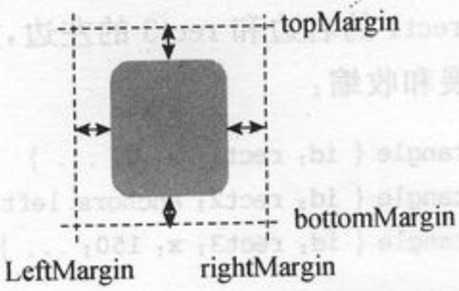



图 6-15　项目锚边距示意图

图 6-14 中没有显示 baseline，它是一条假想的线，文本坐落在这条线上。对于没有文本的项目，它与 top 相同。编程中还经常用到 anchors.fill 来将一个项目充满另一个项目，它等价于使用了 4 个直接的锚。但是要注意，只能在父子或者兄弟之间使用锚，而且，基于锚的布局不能和绝对的位置定义(比如直接设置 x 和 y 属性的值)混合使用，否则会出现不确定的结果。对应本节的内容，可以在 Qt 帮助中查看 Anchor-based Layout in QML 关键字。

来看几个例子。在下面的代码中，将一个文本显示在了一张图片下面，并使其水平居中显示：

```
Image { id: pic; ... }
```

```
Text {
  id: label

  anchors.horizontalCenter: pic.horizontalCenter
  anchors.top: pic.bottom
  anchors.topMargin: 5
  ...
}
```

代码的运行效果示意图如图6-16(a)所示。下面让文本在图片的右侧显示,它们的y属性默认都为0:

```
Image { id: pic; ... }

Text {
  id: label

  anchors.left: pic.right
  anchors.leftMargin: 5
  ...
}
```

代码的运行效果示意图如图6-16(b)所示。

通过指定多个水平和垂直的锚还可以控制项目的大小。在下面的代码中,rect2锚定到rect1的右边和rect3的左边,如果移动rect1或者rect3,那么rect2会进行必要的伸展和收缩:

```
Rectangle { id: rect1; x: 0; ... }
Rectangle { id: rect2; anchors.left: rect1.right; anchors.right: rect3.left; ... }
Rectangle { id: rect3; x: 150; ... }
```

代码的运行效果示意图如图6-17所示。从这几个例子可以看到,使用锚是进行相对位置布局的,而且还可以根据相关项目的位置和大小的改变而自动进行其他项目位置和大小的改变,这在实际界面编程中是很有用的,所以建议在QML中多使用锚来进行布局。

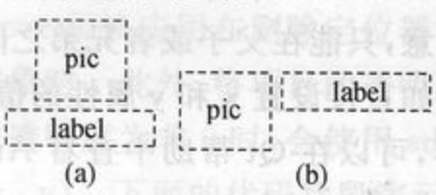



图6-16 使用锚部件示意图

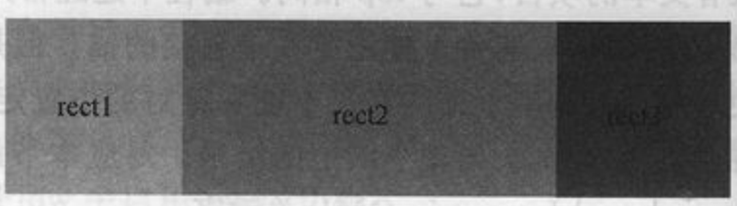



图6-17 使用锚限定项目的大小示意图

# 6.4 基本可视元素

## 6.4.1 Item

QML 中的所有可视项目都继承自 Item。虽然 Item 本身没有可视化的外观，但是它定义了可视化项目的所有属性，例如关于位置的 x 和 y 属性、关于大小的 width 和 height 属性、关于布局的 anchors 相关属性以及关于按键处理的 Keys 属性等。Item拥有一个 visible 属性，将其设置为 false 可以隐藏项目，该属性默认值为 true。下面介绍几点与 Item 相关的内容，而与 Item 相关的更多的其他内容，比如按键处理、状态切换等将会在后面的章节中讲到。

### 1. 作为容器

Item 也常用来对项目进行分组，例如：

```
Item {
    Image {
        source: "tile.png"
    }
    Image {
        x: 80
        width: 100
        height: 100
        source: "tile.png"
    }
    Image {
        x: 190
        width: 100
        height: 100
        fillMode: Image.Tile
        source: "tile.png"
    }
}
```

### 2. 默认属性

Item 中有一个 children 属性和一个 resources 属性，前者包含了可见的孩子的列表，后者包含了不可见的资源。例如：

```
Item {
    children: [
        Text {},
        Rectangle {}
```

```
    ]
    resources: [
      Timer {}
    ]
}
```

Item中还有一个data默认属性，它允许在一个项目中将可见的孩子和不可见的资源自由混合。也就是说，如果向data列表中添加一个可视项目，那么将作为一个孩子进行添加；而如果添加任何其他的对象类型，那么将被作为资源进行添加。又因为data是默认属性，所以可以省略data标签，这样前面的代码可以改写为：

```
Item {
    Text {}
    Rectangle {}
    Timer {}
}
```

这也就是说，编程中可以直接向一个项目中添加任何的元素。

**3. 透明度**

Item有一个opacity属性，可以用来设置透明度。该属性可以设置为在0.0(完全透明)和1.0(完全不透明)之间的数字，默认值为1.0。opacity是一个继承属性，也就是说，透明度也会应用到子项目上。在大多数情况下会产生想要的结果，不过有些时候，会产生意外的结果。例如下面的代码是两个相交的不透明的矩形。(项目源码路径：src\6\6-7\myItem)

```
Item {
    Rectangle {
        color: "red"
        width: 100; height: 100
        Rectangle {
            color: "blue"
            x: 50; y: 50; width: 100; height: 100
        }
    }
}
```

但是下面的代码会使红色和蓝色的矩形都变成透明的：

```
Item {
    Rectangle {
        opacity: 0.5
        color: "red"
        width: 100; height: 100
        Rectangle {
            color: "blue"
```

```
        x: 50; y: 50; width: 100; height: 100
    }
  }
}
```

**4. 堆叠顺序**

Item 拥有一个 z 属性，可以用来设置兄弟项目的堆叠顺序。默认的堆叠顺序为 0。拥有较大 z 属性值的项目会绘制在比其 z 属性值小的兄弟项目之上。拥有相同的 z 属性值的项目会以出现的顺序由下向上绘制。如果项目的 z 属性值为负值，那么会被绘制在父项目的内容的下面。下面来看一些例子。

具有相同的 z 值，那么后面出现的项目会在前面出现项目的上面。例如下面的代码中蓝色矩形在红色矩形上面。（项目源码路径：src\6\6 - 8\myItem）

```
Item {
  Rectangle {
    color: "red"
    width: 100; height: 100
  }
  Rectangle {
    color: "blue"
    x: 50; y: 50; width: 100; height: 100
  }
}
```

如果一个项目的 z 值较大，那么绘制在上面。例如下面的代码中红色矩形绘制在蓝色矩形上面：

```
Item {
  Rectangle {
    z: 1
    color: "red"
    width: 100; height: 100
  }
  Rectangle {
    color: "blue"
    x: 50; y: 50; width: 100; height: 100
  }
}
```

如果具有相同的 z 值，那么子项目会绘制在其父项目上面。例如下面的代码中蓝色矩形绘制在红色矩形上面：

```
Item {
  Rectangle {
    color: "red"
```

```
    width: 100; height: 100
    Rectangle {
      color: "blue"
      x: 50; y: 50; width: 100; height: 100
    }
  }
}
```

如果一个子项目拥有一个负的z值,那么绘制在其父项目下面。例如下面的代码中蓝色矩形被绘制在红色矩形下面:

```
Item {
  Rectangle {
    color: "red"
    width: 100; height: 100
    Rectangle {
      z: -1
      color: "blue"
      x: 50; y: 50; width: 100; height: 100
    }
  }
}
```

**5. 定位子项目和坐标映射**

Item中提供了childAt(real x, real y)函数来返回在点(x, y)处的子项目,如果没有这样的项目则返回null。而mapFromItem(Item item, real x, real y)函数会将item坐标系统中的点(x, y)映射到该项目的坐标系统上,返回一个包含映射后的x和y属性的对象。如果item被指定为null值,那么会从根QML视图坐标系统上的点进行映射。对应的还有一个mapToItem(Item item, real x, real y)函数,它与mapFromItem()是很相似的,只不过是从当前项目坐标系统的(x, y)点映射到item的坐标系统而已。

## 6.4.2 Rectangle

Rectangle项目用来使用纯色或者渐变来填充一个区域,也经常用来存放其他项目。每一个Rectangle项目或者使用color属性指定一个纯色来填充,或者使用gradient属性指定一个Gradient元素定义的渐变来填充。如果既使用了color又使用了gradient,那么最终会使用gradient。可以为一个矩形添加一个可选的边界,并通过border.color和border.width为其指定颜色和宽度。也可以使用radius属性来产生一个圆角的矩形,这样会使矩形的角产生弧形边缘,为了使其更为圆滑,可以指定smooth属性。使用smooth属性可以提高外观表现,不过这是以损失渲染性能为代价的,所以建议在运动的矩形中不要设置该属性,而只在静态的矩形中进行设置。

下面的代码展示了一个红色的圆角矩形，其运行效果如图 6－18 所示。（项目源码路径：src\6\6－9\myRectangle）

```
import QtQuick 1.0

Rectangle {
    width: 100
    height: 100
    color: "red"
    border.color: "black"
    border.width: 5
    radius: 10
}
```

### 6.4.3　Text

一个 Text 项目可以显示纯文本或者富文本。例如下面代码的效果如图 6－19 所示。（项目源码路径：src\6\6－10\myText）

图 6－18　Rectangle 运行效果


图 6－19　Text 运行效果

```
import QtQuick 1.0

Column {
    Text { text: "Hello World!"; font.family: "Helvetica";
        font.pointSize: 24; color: "red" }
    Text { text: "<b>Hello</b> <i>World! </i>" }
}
```

如果没有明确指定高度和宽度，Text 会尝试确定需要多大的空间并依此来设置。如果没有使用 wrapMode 属性设置换行，那么所有的文本将会放置在单行上。而对于设置了宽度并且只想在单行中显示纯文本，那么可以使用 elide 属性，它可以为超出宽度的文本提供自动省略（使用“…”来表示省略）。注意，Text 支持的 HTML 子集是有限的，并且，如果在文本中包含 HTML 的 img 标签来加载远程的图片，那么文本会被重载。Text 提供了只读的文本，如果要使用可编辑的文本，可以使用后面讲到的 TextEdit 元素。

### 1. 文本省略

Text 中的 elide 属性可以设置省略文本的部分内容来适合 Text 项目的宽度。如果没有对 Text 明确设置 width 值，那么 elide 属性将不起作用。这个属性也无法使用多行文本和富文本。elide 属性可取的值有 Text. ElideNone（默认）、Text. ElideLeft、Text. ElideMiddle 和 Text. ElideRight。例如下面的代码的运行效果如图 6 - 20 所示，如果中文显示为乱码，可以在“编辑→选择编码”菜单项中选择 UTF - 8 编码保存。（项目源码路径：src\6\6 - 11\myText）

```
import QtQuick 1.0

Column {
  spacing: 20

  Text {
    width: 200;
    text: "使文本在单行中对于超出部分不要进行省略"
  }
  Text {
    width: 200; elide: Text.ElideLeft;
    text: "使文本在单行中对于超出部分从左边进行省略"
  }
  Text {
    width: 200; elide: Text.ElideMiddle;
    text: "使文本在单行中对于超出部分从中间进行省略"
  }
  Text {
    width: 200; elide: Text.ElideRight;
    text: "使文本在单行中对于超出部分从右边进行省略"
  }
}
```

使文本在单行中对于超出部分不要进行

…行中对于超出部分从左边进行省略

使文本在单行中…从中间进行省略

使文本在单行中对于超出部分从右…

图 6 - 20　文本省略运行效果

### 2. 字　体

在 Text 中使用 font 分组属性来对字体进行设置，如表 6 - 3 所列。

表 6-3　Text 中 font 属性

| 属　性 | 作　用 | 值 |
| --- | --- | --- |
| font. bold | 是否加粗 | true 或者 false |
| font. capitalization | 大写策略 | Font. MixedCase:不改变大小写(默认值)<br>Font. AllUppercase:全部大写<br>Font. AllLowercase:全部小写<br>Font. SmallCaps:小型大写字母(即小写字母变为大写但不改变大小)<br>Font. Capitalize:将首字母大写 |
| font. family | 字体族 | 字体族的名字(区分大小写) |
| font. italic | 是否斜体 | true 或者 false |
| font. letterSpacing | 字符间距 | 正值加大间距,负值减小间距 |
| font. pixelSize | 字体大小 | 整数(单位为像素,依赖于设备) |
| font. pointSize | 字体大小 | 大于 0 的值(是设备无关的) |
| font. strikeout | 是否有删除线 | true 或者 false |
| font. underline | 是否有下划线 | true 或者 false |
| font. weight | 字体重量 | Font. Light、Font. Normal(默认)、Font. DemiBold、Font. Bold、Font. Black |
| font. wordSpacing | 单词间距 | 正值加大间距,负值减小间距 |

QML 中还提供了 FontLoader 元素来加载字体,这里可以指定一个字体的名称,或者是一个 URL,也就是说可以指定网络上的一个字体文件。例如:

```
import QtQuick 1.0

Column {
  FontLoader { id: fixedFont; name: "Courier" }
  FontLoader { id: webFont; source: "http://www.mysite.com/myfont.ttf" }
  Text { text: "Fixed - size font"; font.family: fixedFont.name }
  Text { text: "Fancy font"; font.family: webFont.name }
}
```

### 3. 对齐方式

Text 中的 horizontalAlignment 和 verticalAlignment 分别用来设置文本在 Text 项目区域中的水平对齐方式和垂直对齐方式。默认的,文本在左上方。对于水平对齐方式有 Text. AlignLeft、Text. AlignRight 和 Text. AlignHCenter;而垂直对齐方式有 Text. AlignTop、Text. AlignBottom 和 Text. AlignVCenter。例如下面代码的

运行效果如图6-21所示。(项目源码路径:src\6\6-12\myText)

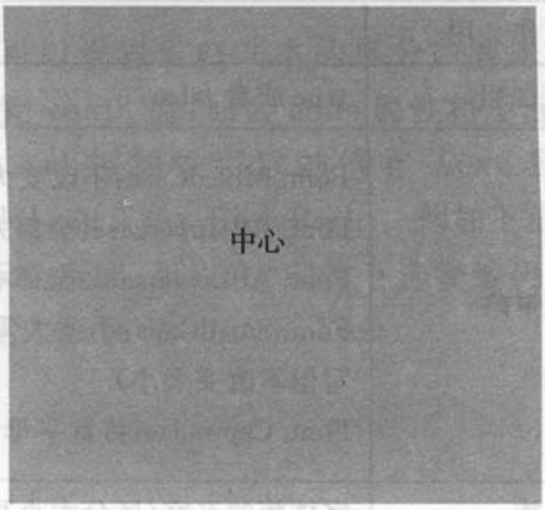



图6-21 文本对齐方式

```
import QtQuick 1.0

Rectangle {
  width: 200; height: 200; color: "red"

  Text {
    width: 200; height: 200
    horizontalAlignment: Text.AlignHCenter
    verticalAlignment: Text.AlignVCenter
    text: "中心"
  }
}
```

对于没有设置Text大小的单行文本,Text的大小就是包含文本的区域。在这种情况下,所有的对齐都是等价的。如果想让文本处于父项目的中间,那么可以使用Item::anchors属性来实现。

**4. 文本样式**

使用Text的style属性可以来设置文本的样式,支持的文本样式有Text.Normal、Text.Outline、Text.Raised和Text.Sunken。例如下面的代码的运行效果如图6-22所示。(项目源码路径:src\6\6-13\myText)

```
import QtQuick 1.0

Row {
  Text { font.pointSize: 24; text: "Normal" }
  Text { font.pointSize: 24; text: "Raised"; style: Text.Raised;
    styleColor: "#AAAAAA" }
  Text { font.pointSize: 24; text: "Outline";style: Text.Outline;
    styleColor: "red" }
```

```
    Text { font.pointSize: 24; text: "Sunken"; style: Text.Sunken;
      styleColor: "#AAAAAA" }
  }
```


图 6-22　文本样式运行效果

**5. 文本换行**

通过设置 Text 的 wrapMode 属性可以实现文本的自动换行。只有在明确设置了 Text 的 width 属性时换行才会起作用。可用的换行模式有：

- Text. NoWrap(默认)：不进行换行；
- Text. WordWrap：在单词边界进行换行；
- Text. WrapAnywhere：只要达到边界，则会在任意点进行换行，甚至是在一个单词的中间；
- Text. Wrap：如果可能，会尽量在单词边界进行换行，否则，会在任意点换行。

**6. 链接信号**

Text 中提供了一个 Text::onLinkActivated(string link)处理器，它会在用户单击文本中嵌入的一个链接时被调用。链接必须在富文本或者 HTML 格式中，而函数中 link 字符串提供了被单击的可以访问的特定链接。如下例：(项目源码路径：src\6\6-14\myText)

```
Text {
    textFormat: Text.RichText
    text: "The main website is at <a href = \"http://qt.nokia.com\">Nokia Qt DF</
a>."
    onLinkActivated: console.log(link + " link activated")
}
```

这里显示的文本中“Nokia Qt DF”为链接文本，单击它，则输出“http://qt.nokia.com link activated”字符串。

## 6.4.4　TextEdit

TextEdit 项目用来显示一块可编辑的、格式化的文本。TextEdit 与 Qt 中的 QTextEdit 很相似，既可以显示纯文本也可以显示富文本。如下例：(项目源码路径：src\6\6-15\myTextEdit)

```
TextEdit {
    width: 240
    text: "<b>Hello</b> <i>World! </i>"
```

```
    font.family: "Helvetica"
    font.pointSize: 20
    color: "blue"
    focus: true
  }
```

这里将 focus 属性设置为 true,这样可以使 TextEdit 项目来接收键盘焦点。注意,TextEdit 没有为用户界面提供滚动、光标跟随或者其他的行为。一般是使用 Flickable 元素来提供滚动实现光标跟随。如下例:(项目源码路径:src\6\6-16\myTextEdit)

```
import QtQuick 1.0

Flickable {
  id: flick

  width: 300; height: 200;
  contentWidth: edit.paintedWidth
  contentHeight: edit.paintedHeight
  clip: true

  function ensureVisible(r)
  {
    if (contentX >= r.x)
      contentX = r.x;
    else if (contentX + width <= r.x + r.width)
      contentX = r.x + r.width - width;
    if (contentY >= r.y)
      contentY = r.y;
    else if (contentY + height <= r.y + r.height)
      contentY = r.y + r.height - height;
  }

  TextEdit {
    id: edit
    width: flick.width
    height: flick.height
    focus: true
    wrapMode: TextEdit.Wrap
    onCursorRectangleChanged: flick.ensureVisible(cursorRectangle)
  }
}
```

这里的 Flickable 元素会在 6.6.6 小节讲到。TextEdit 中通过 cut()、copy()和 paste()等函数提供了对剪贴板的支持。通过设置 selectByMouse 属性可以启用鼠

标进行选取内容的功能。也可以完全使用 QML 代码来完成内容的选取，这需要设置 selectionStart 和 selectionEnd，然后使用 selectAll()或者 selectWord()函数进行操作。这个元素中还包含了 openSoftwareInput Panel() 和 closeSoftwareInput Panel()函数分别用来在特定的设备上打开和关闭软键盘。TextEdit 的字体、换行等属性与 Text 元素相似，这里就不再进行介绍。关于这个元素，可以参考 Qt 提供的 Text Selection example 示例程序。

### 6.4.5　TextInput

TextInput 元素用来显示单行的可编辑的纯文本。TextInput 与 Qt 中的 QLineEdit 很相似，用来接收一行文本输入。在一个 TextInput 项目上可以使用输入限制，例如使用验证器或者输入掩码。而通过设置 echoMode 可以使得 TextInput 用来输入密码。下面的代码中演示了使用整型验证器 IntValidator 来实现在 TextInput 中只能输入 11～31 之间的整数。(项目源码路径：src\6\6 - 17\myTextInput)

```
import QtQuick 1.0

TextInput{
  validator: IntValidator{ bottom: 11; top: 31; }
  focus: true
}
```

可用的验证器还有 DoubleValidator(非整数验证器)和 RegExpValidator(正则表达式验证器)。对于正则表达式的使用，可以参考《Qt Creator 快速入门》中第 7 章的相关内容。TextEdit 中也可以使用输入掩码来限制输入的内容，关于输入掩码的使用，可以参考《Qt Creator 快速入门》中第 3 章 QLineEdit 部分的内容。

## 6.5　事件处理

在以前讲解 Qt C++编程时就讲到了事件的处理，比如鼠标事件、键盘事件等。在 QML 编程中同样需要对鼠标键盘等事件进行处理。因为 QML 程序更多的是实现触摸式用户界面，所以更多的是对鼠标(在触屏设备上可能是手指)单击的处理。与以前的窗口部件不同，在 QML 中如果一个项目想要能够被单击，就要在其上放置一个 MouseArea 元素，也就是说，用户只能在 MouseArea 确定的范围内进行单击。

### 6.5.1　MouseArea

MouseArea 是一个不可见的项目，通常用来和一个可见的项目配合使用来为其提供鼠标处理。鼠标处理的逻辑可以包含在一个 MouseArea 项目中。MouseArea

的enabled属性可以用来设置是否启用鼠标处理，值默认为true，如果设置为false，MouseArea对于鼠标事件将会变为透明的，也就是说不再处理任何鼠标事件。而只读的pressed属性表明了是否用户在MouseArea按住了鼠标按钮，这个属性经常用于属性绑定，可以实现在鼠标按下时执行一些操作。只读的containsMouse属性表明了当前是否有鼠标光标在MouseArea上，默认的，只有鼠标的一个按钮处于按下状态才可以被检测到。对于鼠标位置和按钮单击等信息是通过信号提供的，可以使用事件处理器来获取，最常用的有onClicked()、onDoubleClicked()、onPressed()、onReleased()和onPressAndHold()等。默认情况下，MouseArea项目只报告鼠标单击而不报告鼠标光标的位置改变，这个可以通过设置hoverEnabled属性来进行更改。这样onPositionChanged()、onEntered()和onExited()等处理函数都可以使用了，而且这时containsMouse属性也可以在没有鼠标按钮按下的情况下进行检查了。

下面来看一个简单的例子，这里初始是一个绿色的正方形，当在其上单击以后就会变为红色。代码如下：(项目源码路径：src\6\6-18\myMouseArea)

```
import QtQuick 1.0

Rectangle {
  width: 100; height: 100
  color: "green"

  MouseArea {
    anchors.fill: parent
    onClicked: { parent.color = 'red' }
  }
}
```

这里使用了anchors.fill: parent来使MouseArea填充满整个Rectangle区域，因为只有在MouseArea上单击才能进行处理，现在MouseArea覆盖了整个Rectangle，所以在Rectangle的任何位置单击都有效果。

**1. MouseEvent**

在很多MouseArea的信号中都包含了一个鼠标事件参数，例如MouseArea::onClicked(MouseEvent mouse)。在MouseEvent对象中，可以通过x和y属性获取鼠标的位置；通过button属性可以获取按下的按键；通过modifiers属性可以获取按下的键盘修饰符等。这里的button可取的值有Qt.LeftButton左键、Qt.RightButton右键和Qt.MiddleButton中键；而modifiers的值由多个按键进行位组合而成，在使用时需要将modifiers与这些特殊的按键进行按位与来判断按键，常用的按键有Qt.NoModifier没有修饰键、Qt.ShiftModifier一个Shift按键、Qt.ControlModifier一个Ctrl按键、Qt.AltModifier一个Alt按键。下面来看一个例子。(项目源码路径：src\6\6-19\myMouseEvent)

```
import QtQuick 1.0

Rectangle {
  width: 100; height: 100
  color: "green"

  MouseArea {
    anchors.fill: parent
    acceptedButtons: Qt.LeftButton | Qt.RightButton
    onClicked: {
      if (mouse.button == = Qt.RightButton)
        parent.color = 'blue';
      else if ((mouse.button == = Qt.LeftButton)
          && (mouse.modifiers & Qt.ShiftModifier))
        parent.color = 'green';
      else
        parent.color = 'red';
    }
  }
}
```

这里实现的功能是，当按下鼠标右键时，Rectangle 变为蓝色，当按下鼠标左键时，变为红色，而当在按着键盘上的 Shift 键的同时按下鼠标左键，那么重新变为绿色。

**2. 拖　拽**

MouseArea 中的 drag 分组属性提供了一个使项目可以拖拽的简便方法。其中，drag. target 属性用来指定拖动的项目的 id；drag. active 属性获取项目当前是否正在被拖动的信息；drag. axis 属性用来指定属性拖动的方向，可以是水平方向(Drag. XAxis)、垂直方向(Drag. YAxis)或者两个方向都可以(Drag. XandYAxis)；drag. minimum 和 drag. maximum 限制了项目在指定方向上拖动的距离。如下例。(项目源码路径：src\6\6 - 20\myDrag)

```
import QtQuick 1.0

Rectangle {
  id: container
  width: 600; height: 200
  Rectangle {
    id: rect
    width: 50; height: 50
    color: "red"
    opacity: (600.0 - rect.x) / 600
```

```
        MouseArea {
            anchors.fill: parent
            drag.target: rect
            drag.axis: Drag.XAxis
            drag.minimumX: 0
            drag.maximumX: container.width - rect.width
        }
    }
}
```

这样将可以将红色小方块水平进行拖动，并且在拖动的过程中，小方块的透明度是根据距离而变化的。

### 6.5.2 按键处理

当一个按键按下或者释放时，则产生一个键盘事件，并将其传递给获得有焦点的QML项目(如果将一个项目的focus属性(Item元素的属性)设置为true，那么这个项目便会获得焦点)。为了方便创建可重用的组件并解决一些实现流程的用户界面的独特问题，QML项目在Qt传统的键盘焦点模型上添加了基于作用域的扩展。对应本节内容，可以在Qt帮助中参考Keyboard Focus in QML关键字。

**1. 按键处理概述**

当用户按下或者释放一个按键，会按以下步骤进行处理：

第一步，Qt获取键盘动作并产生一个键盘事件。

第二步，如果包含QDeclarativeView的Qt部件具有焦点，那么键盘事件会传递给它，否则将进行常规的按键处理。

第三步，场景将键盘事件交付给具有活动焦点的QML项目。如果没有项目具有活动焦点，键盘事件会被忽略，然后继续常规的按键处理。

第四步，如果具有活动焦点的QML项目接受了该键盘事件，那么传播将停止。否则，该事件会递归的传递到每一个项目的父项目，直到被接受，或者到达根项目。

第五步，如果到达了根项目，该键盘事件会被忽略而继续常规的Qt按键处理。

所有的基于Item的可见元素都可以通过Keys附加属性来进行按键处理。Keys附加属性提供了基本的处理器，例如onPressed和onReleased，也提供了对特殊按键的处理器，例如onCancelPressed。例如下面的代码实现了按下A键输出提示字符串：(项目源码路径：src\6\6-21\myKeyEvent)

```
import QtQuick 1.0

Rectangle {
    width: 100; height: 100
    focus: true
```

```
    Keys.onPressed: {
        if (event.key == Qt.Key_A) {
            console.log("Key A was pressed");
            event.accepted = true;
        }
    }
}
```

这里的 event.accepted 设置为 true 可以防止事件向上层项目传播。可以参考 Keys 元素的帮助文档来查看其提供的所有处理器。在这些处理器中大多含有一个 KeyEvent 元素，它提供了关于该键盘事件的信息。例如这里的 event.key 就是获取按下的按键，另外它还有 accepted、isAutoRepeat、modifiers 和 text 等属性。

QML 中还有一个 KeyNavigation 元素，它也是一个附加属性，可以用来实现使用方向键或者 Tab 键来进行项目的导航。它的属性有 backtab、down、left、priority、right、tab 和 up 等。例如，下面的代码实现了使用方向键在 2×2 的项目网格中进行导航。（项目源码路径：src\6\6-22\myKeyNavigation）

```
import QtQuick 1.0

Grid {
    width: 100; height: 100
    columns: 2

    Rectangle {
        id: topLeft
        width: 50; height: 50
        color: focus ? "red" : "lightgray"
        focus: true
        KeyNavigation.right: topRight
        KeyNavigation.down: bottomLeft
    }

    Rectangle {
        id: topRight
        width: 50; height: 50
        color: focus ? "red" : "lightgray"
        KeyNavigation.left: topLeft
        KeyNavigation.down: bottomRight
    }

    Rectangle {
        id: bottomLeft
        width: 50; height: 50
        color: focus ? "red" : "lightgray"
```

```
        KeyNavigation.right: bottomRight
        KeyNavigation.up: topLeft
    }

    Rectangle {
        id: bottomRight
        width: 50; height: 50
        color: focus ? "red" : "lightgray"
        KeyNavigation.left: bottomLeft
        KeyNavigation.up: topRight
    }
}
```

**2. 查询活动焦点项目**

一个项目是否具有活动焦点，可以通过 Item::activeFocus 属性进行查询。例如，下面代码中 Text 元素的文本就决定于它是否获取了焦点：

```
Text {
    text: activeFocus ? "I have active focus!" : "I do not have active focus"
}
```

**3. 获取焦点和焦点作用域**

一个项目可以通过设置其 focus 属性为 true 来使其获得焦点。一般情况下，使用 focus 属性就足够了，例如下面是 MyWidget.qml 文件中的内容，它只是简单地将 focus 设置为了 true，从而获得了焦点。

```
import QtQuick 1.0

Rectangle {
    id: widget
    color: "lightsteelblue"; width: 175; height: 25; radius: 10; smooth: true
    Text { id: label; anchors.centerIn: parent}
    focus: true
    Keys.onPressed: {
        if (event.key = = Qt.Key_A)
            label.text = 'Key A was pressed'
        else if (event.key = = Qt.Key_B)
            label.text = 'Key B was pressed'
        else if (event.key = = Qt.Key_C)
            label.text = 'Key C was pressed'
    }
}
```

然而，当将上面代码作为一个可重用或者可被导入的组件时，简单地使用 focus

属性就不再有效。作为演示，在下面的代码中创建了 MyWidget.qml 组件的两个实例，然后将第一个设置为具有焦点，这样的目的是当按下 A、B 或者 C 键时第一个实例可以接收到事件并做出相应的响应，下面是 window.qml 文件的内容。（项目源码路径：src\6\6-23\window）

```
import QtQuick 1.0

Rectangle {
  id: window
  color: "white"; width: 240; height: 150

  Column {
    anchors.centerIn: parent; spacing: 15

    MyWidget {
      focus: true
      color: "lightblue"
    }
    MyWidget {
      color: "palegreen"
    }
  }
}
```

我们希望通过为第一个 MyWidget 对象的 focus 属性设置为 true 来使其获得焦点，然而，当运行代码后，会发现是第二个对象获得了焦点。现在观察 MyWidget 和 window 的代码，会发现问题是很明显的：这里有三个元素设置了 focus 属性为 true：两个 MyWidget 设置了 focus 为 true 以及 window 组件中也设置了 focus。最终，只有一个元素可以获得键盘焦点，这样系统可以决定哪个元素接收焦点。当创建了第二个 MyWidget 时，因为它是最后一个设置 focus 属性为 true 的元素，所以它将接收焦点。是因为不可见的因素造成了这个问题，MyWidget 组件希望获得焦点，但是当它被导入或者被重用后它将无法控制焦点。也就是说，window 组件无法知道它导入的组件是否请求了焦点。为了解决这个问题，在 QML 中引入了所谓的焦点作用域(focus scope)的概念，一个焦点作用域通过 FocusScope 元素来创建。例如，下面在 MyWidget.qml 组件中使用 FocusScope 元素。（项目源码路径：src\6\6-24\window）

```
import QtQuick 1.0

FocusScope {
  // FocusScope 需要绑定到子项目的可视属性上
  property alias color: rectangle.color
```

```
    x: rectangle.x; y: rectangle.y
    width: rectangle.width; height: rectangle.height

    Rectangle {
      id: rectangle
      anchors.centerIn: parent
      color: "lightsteelblue"; width: 175; height: 25; radius: 10; smooth: true
      Text { id: label; anchors.centerIn: parent }
      focus: true
      Keys.onPressed: {
        if (event.key == Qt.Key_A)
          label.text = "Key A was pressed"
        else if (event.key == Qt.Key_B)
          label.text = "Key B was pressed"
        else if (event.key == Qt.Key_C)
          label.text = "Key C was pressed"
      }
    }
}
```

现在运行 window.qml 的代码就可以显示正确的结果了。注意，因为 FocusScope 元素不是可视化元素，需要将它的子项目的属性暴露给它的父项目。布局和定位元素可以使用这些可见和样式属性来创建布局。在这个例子中，如果没有将 Rectangle 的属性直接绑定到 FocusScope 上，那么 Column 元素就无法正确的显示两个部件。

### 6.5.3 定时器

定时器可以用来将一个动作在指定的时间间隔触发一次或者多次，在 QML 中使用 Timer 元素来表示一个定时器。下面的代码中使用了一个定时器来显示当前的日期和时间，并每隔 500 毫秒更新一次文本的显示。这里使用了 JavaScript 的 Date 对象来获取当前的时间。（项目源码路径：src\6\6-25\myTimer）

```
import QtQuick 1.0

Item {
  Timer {
    interval: 500; running: true; repeat: true
    onTriggered: time.text = Date().toString()
  }

  Text { id: time }
}
```

这里的 interval 属性用来设置时间间隔，单位是毫秒，默认值是 1000 毫秒；repeat 属性用来设置是否重复触发，如果为 false，则只触发一次并自动将 running 属性设置为 false，其默认值为 false；当将 running 属性设置为 true 时将开启定时器，否则停止定时器，其默认值为 false；当定时器触发时会执行 onTriggered()信号处理函数，这里可以定义一些操作；Timer 还提供了一些函数，比如 restart()、start()和 stop()等。

注意，如果定时器正在运行的过程中它的一个属性被更改，那么经过的时间将被重置。例如，如果一个间隔为 1000 毫秒的定时器在经过了 500 毫秒以后，它的 repeat 属性被改变了，那么经过的时间将会被重置为 0，再过 1000 毫秒以后才会触发定时器。关于定时器的使用，可以参考 Qt 自带的 Clocks Example 示例程序。

### 6.5.4　使用 Loader 动态加载组件

Loader 元素用来动态加载可见的 QML 组件，它可以加载一个 QML 文件(使用 source 属性)或者一个组件对象(使用 sourceComponent 属性)。这个对于拖延组件的创建是很有用的：例如，当一个组件需要在要求的时候被创建，或者当由于性能原因一个组件不应该被创建时。在下面的代码中当在 MouseArea 单击时加载“Page1.qml”作为一个组件：

```
import QtQuick 1.0

Item {
  width: 200; height: 200

  Loader { id: pageLoader }

  MouseArea {
    anchors.fill: parent
    onClicked: pageLoader.source = "Page1.qml"
  }
}
```

可以通过 item 属性来访问被加载的项目。如果 source 或者 sourceComponent 更改了，任何先前实例化的项目都将被销毁。将 source 设置为空字符串或者将 sourceComponent 设置为 undefined，则销毁当前加载的项目，释放资源并且将 Loader 设置为空。

**1. Loader 的大小行为**

Loader 和其他任何可见的项目一样，必须对其进行位置和大小的设置，这样它才能成为可见的。如果没有明确指定 Loader 的大小，那么 Loader 将会在组件加载完成后自动设置为组件的大小；如果通过设置 width、height 或者使用锚明确指定了

Loader的大小，那么被加载的项目将会改变为Loader的大小。

**2. 从加载的项目中接收信号**

任何从被加载的项目中发射的信号都可以使用Connections元素进行接收。例如，下面的application.qml加载了MyItem.qml，然后通过一个Connections对象来接收加载的项目的message信号。（项目源码路径：src\6\6-6\application）

下面是application.qml文件的内容。

```
import QtQuick 1.0

Item {
  width: 100; height: 100

  Loader {
    id: myLoader
    source: "MyItem.qml"
  }

  Connections {
    target: myLoader.item
    onMessage: console.log(msg)
  }
}
```

下面是MyItem.qml文件的内容：

```
import QtQuick 1.0

Rectangle {
  id: myItem
  signal message(string msg)

  width: 100; height: 100

  MouseArea {
    anchors.fill: parent
    onClicked: myItem.message("clicked!")
  }
}
```

另外，因为MyItem.qml是在Loader的作用域中被加载的，所以可以直接调用在Loader或者其父项目中定义的任何函数。

**3. 焦点和键盘事件**

Loader是一个焦点作用域，对于它的任何获得活动焦点的子项目都必须将fo-

cus 属性设置为 true。任何在被加载的项目中的键盘事件也应该被接受，从而使它们不传播到 Loader。例如，在下面 application.qml 中的 MouseArea 上单击时会加载 KeyReader.qml，注意 Loader 中以及被加载的对象中都将 focus 属性设置为了 true。（项目源码路径：src\6\6-27\application）

下面是 application.qml 文件的内容：

```
import QtQuick 1.0

Rectangle {
  width: 200; height: 200

  Loader {
    id: loader
    focus: true
  }

  MouseArea {
    anchors.fill: parent
    onClicked: loader.source = "KeyReader.qml"
  }

  Keys.onPressed: {
    console.log("Captured:", event.text);
  }
}
```

下面是 KeyReader.qml 文件的内容：

```
import QtQuick 1.0

Item {
  Item {
    focus: true
    Keys.onPressed: {
      console.log("Loaded item captured:", event.text);
      event.accepted = true;
    }
  }
}
```

一旦 KeyReader.qml 被加载完成，它便会接受键盘事件而且这里将 event.accepted 设置为了 true，这样事件就不会传播到父项目 Rectangle 中。

## 6.6 图像、状态和动画

### 6.6.1 渐 变

在QML中使用Gradient项目来定义一个渐变。渐变中的颜色使用一组GradientStop子项目进行定义，它们每一个都在渐变中定义了一个从0.0～1.0之间的位置和一个颜色。每一个GradientStop都可以通过position属性来设置位置，通过color属性来设置颜色。注意，Gradient并不是可见的，需要在一个可见的项目(例如Rectangle)中来使用渐变。例如，下面的代码中定义了一个使用渐变的Rectangle，渐变颜色从红色开始，然后在矩形的1/3高度时变为黄色，最后以绿色结尾。(项目源码路径：src\6\6-28\myGradient)

```
import QtQuick 1.0

Rectangle {
  width: 100; height: 100
  gradient: Gradient {
    GradientStop { position: 0.0; color: "red" }
    GradientStop { position: 0.33; color: "yellow" }
    GradientStop { position: 1.0; color: "green" }
  }
}
```

在Qt 4.7中，只有垂直线性渐变可以使用，如果需要使用不同方向的渐变，可以结合旋转和裁剪操作来实现。

### 6.6.2 图片、边界图片和动态图片

#### 1. 图片 Image

QML中的Image元素用来在声明式用户界面中显示图片。图片资源使用source属性作为一个URL来指定，这里可以应用Qt支持的所有格式，包括PNG、JPEG和SVG等。如果width和height属性没有被指定，那么Image元素自动使用加载的图片的大小。默认的，如果指定了Image的大小，那么图片会缩放到这个大小。这个行为也可以通过设置fillMode属性来改变，它允许图片进行拉伸或者平铺。例如下面的代码中使用平铺方式来显示图片：

```
Image {
  width: 120; height: 120
  fillMode: Image.Tile
```

```
    source: "qtlogo.png"
}
```

所有的填充模式及其效果如图 6 - 23 所示。可以参考 Qt 自带的 Image Example 示例程序。

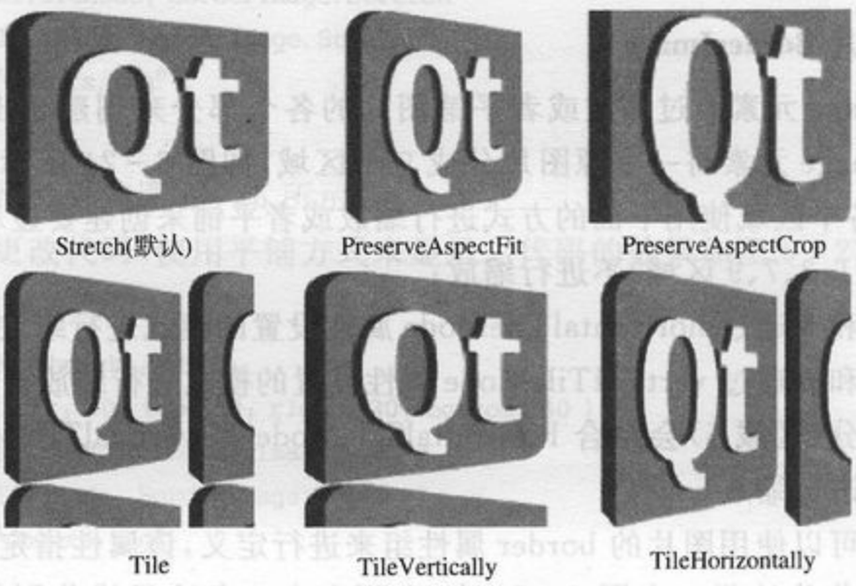



图 6 - 23　图片填充模式效果

默认的，本地的图片会被立即加载，而且在加载完成以前用户界面会被阻塞。如果要加载一个巨大的图片，最好是在一个低优先级的线程中进行，这可以通过将 asynchronous 属性设置为 true 来实现。如果图片需要从网络上获取而不是本地资源，那么它会自动在其他线程中异步加载，progress 和 status 属性会在合适的时间进行更新。图片会在内部进行缓存和共享，所以，如果在几个 Image 元素中使用了相同的 source，那么只会加载该图片的一个备份。注意，图片一般是 QML 用户界面中最占内存的，所以建议将不是界面的组成部分的图片使用 sourceSize 属性设置其大小。这个 sourceSize 属性可以设置 sourceSize. width 和 sourceSize. height，它们与 width 和 height 属性不同，因为设置 Image 的 width 和 height 属性会在绘制图片时进行缩放，而这个属性会设置加载的图片保存时的真实像素数量，这样巨大的图片也不会使用太多的内存了。

在下面的代码中加载了百度主页的 Logo 图片，并输出了加载状态。（项目源码路径：src\6\6 - 29\myImage）

```
import QtQuick 1.0

Image {
    id: image
    width: 120; height: 120
    fillMode: Image.Tile
    source: "http://www.baidu.com/img/baidu_sylogo1.gif"
```

```
    onStatusChanged: {
        if (image.status == Image.Ready) console.log('Loaded')
        else if (image.status == Image.Loading) console.log('Loading')
    }
}
```

**2. 边界图片 BorderImage**

BorderImage 元素通过缩放或者平铺图片的各个部分来创建超出图片的边界。一个 BorderImage 元素将一个源图片分成 9 个区域，如图 6-24 所示。当图片缩放时，源图片的各个区域使用下面的方式进行缩放或者平铺来创建要显示的边界图片：

- 4 个角(1、3、7、9 区域)不进行缩放；
- 区域 2 和 8 通过 horizontalTileMode 属性设置的模式进行缩放；
- 区域 4 和 6 通过 verticalTileMode 属性设置的模式进行缩放；
- 中间部分(区域 5)会结合 horizontalTileMode 和 verticalTileMode 属性设置的模式进行缩放。

这些区域可以使用图片的 border 属性组来进行定义，该属性指定了到源图片每个边缘的距离作为边界。在图 6-24 中，上下左右 4 条边界线分别是 border.top、border.bottom、border.left 和 border.right。可用的平铺模式有 BorderImage.Stretch 拉伸、BorderImage.Repeat 平铺但可能修剪最后的图片、BorderImage.Round 进行平铺但是将图片进行缩小来确保最后的图片不进行修剪。

下面来看一个例子。(项目源码路径：src\6\6-30\myBorderImage)

```
import QtQuick 1.0

Image {
    source: "colors.png"
}
```

这里先使用了 Image 元素来显示原始的图片，效果如图 6-25 所示。下面使用 BorderImage 来显示图片，并将水平方向和垂直方向的平铺模式都设置为拉伸。这样 2、8 区域将会被水平拉伸，4、6 区域会被垂直拉伸。

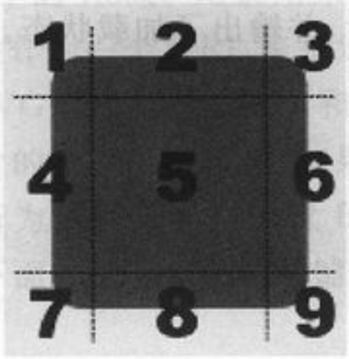



图 6-24　BorderImage 区域示意图

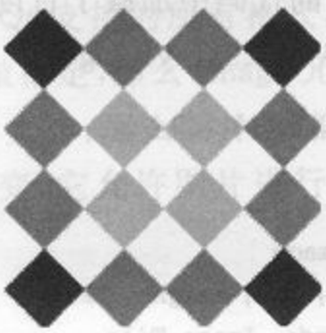


图 6-25　图片没有缩放时效果

```
BorderImage {
    width: 180; height: 180
    border { left: 30; top: 30; right: 30; bottom: 30 }
    horizontalTileMode: BorderImage.Stretch
    verticalTileMode: BorderImage.Stretch
    source: "colors.png"
}
```

代码的运行效果如图 6-26 所示。

下面再次更改代码，使用平铺方式来显示。代码的效果如图 6-27 所示。

```
BorderImage {
    width: 180; height: 180
    border { left: 30; top: 30; right: 30; bottom: 30 }
    horizontalTileMode: BorderImage.Repeat
    verticalTileMode: BorderImage.Repeat
    source: "colors.png"
}
```

关于边界图片的使用可以参考 Qt 提供的 BorderImage example 示例程序。

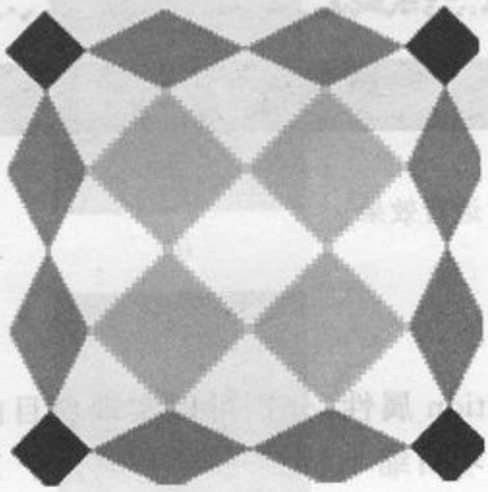


图 6-26　对图片进行拉伸时效果

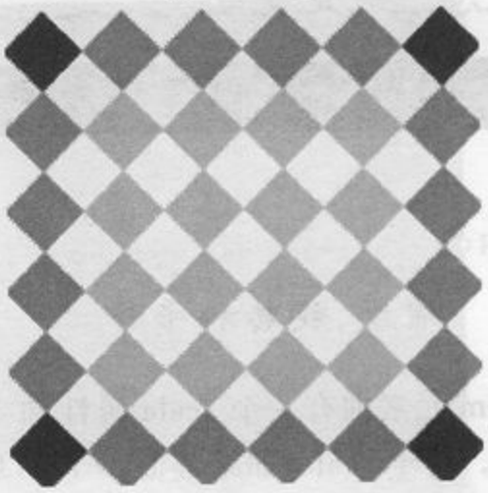


图 6-27　对图片进行平铺时效果

### 3. 动态图片 AnimatedImage

AnimatedImage 扩展了 Image 元素的功能，可以用来播放包含了一系列帧的图片动画，比如 GIF 文件。当前帧和动画总长度等信息可以使用 currentFrame 和 frameCount 属性来获取。可以通过改变 playing 和 paused 属性的值来开始、暂停和停止动画。下面的例子中通过获取动画当前帧和总帧数来实现了播放进度的显示。(项目源码路径：src\6\6-31\myAnimatedImage)

```
import QtQuick 1.0
```

```
Rectangle {
  width: animation.width; height: animation.height + 8

  AnimatedImage { id: animation; source: "animation.gif" }

  Rectangle {
    property int frames: animation.frameCount

    width: 4; height: 8
    x: (animation.width - width) * animation.currentFrame / frames
    y: animation.height
    color: "red"
  }
}
```

代码的运行效果如图 6-28 所示。

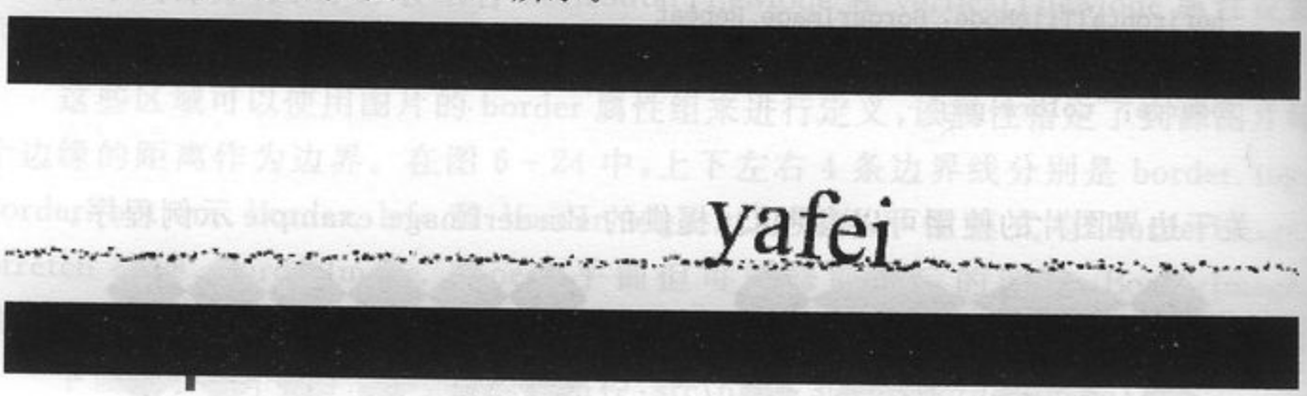



图 6-28 动态图片运行效果

### 6.6.3 缩放、旋转和平移

Item 元素拥有一个 scale 属性和一个 rotation 属性，分别可以实现项目的缩放和旋转。对于 scale，如果其值小于 1，那么会将项目缩小显示；如果其值大于 1，那么会将项目放大显示。如果使用一个负值，那么将会显示镜像效果。scale 的默认值是 1，也就是显示正常大小。例如下面的例子将红色矩形放大了 1.6 倍进行显示。（项目源码路径：src\6\6-32\scaleItem）

```
import QtQuick 1.0

Rectangle {
  color: "blue"
  width: 100; height: 100
  Rectangle {
    color: "green"
    width: 25; height: 25
  }
```

```
    Rectangle {
        color: "red"
        x: 25; y: 25; width: 25; height: 25
        scale: 1.6
    }
}
```

代码的运行效果如图 6-29(a)所示。缩放是以 transformOrigin 属性指定的点为原点进行的，可用的点一共有 9 个，默认的原点是 Center 即项目的中心，如图 6-30 所示。如果想使用任意的点作为原点，那么就要使用后面讲到的 Scale 和 Rotation 元素了。下面将红色矩形的定义代码进行更改，使用 TopLeft 为原点，运行代码效果如图 6-29(b)所示。

```
Rectangle {
    color: "red"
    x: 25; y: 25; width: 25; height: 25
    scale: 1.6
    transformOrigin: "TopLeft"
}
```

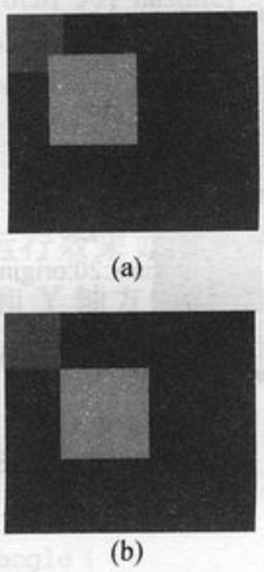



图 6-29　项目缩放效果

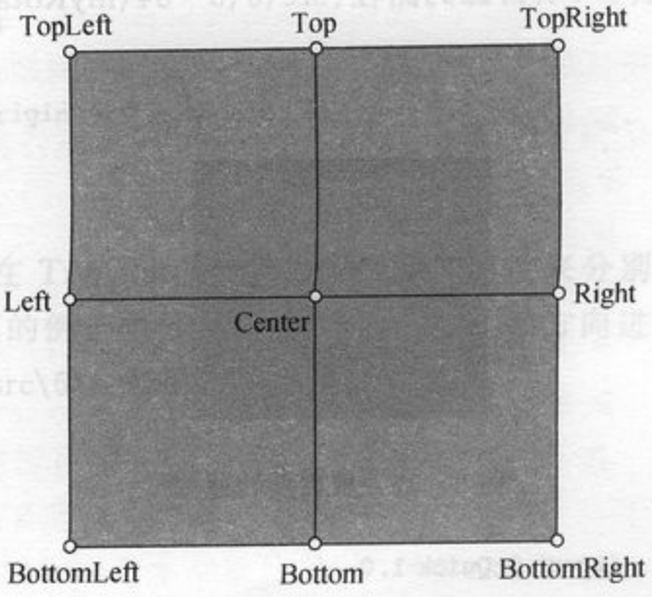



图 6-30　项目变换原点示意图

可以使用 rotation 属性来指定项目顺时针旋转的度数，默认值为 0。旋转也是以 transformOrigin 属性指定的点为原点进行的。例如下面的代码中将红色的矩形顺时针旋转了 30°。(项目源码路径：src\6\6-33\rotationItem)

```
import QtQuick 1.0

Rectangle {
    color: "blue"
    width: 100; height: 100
```

```
    Rectangle {
      color: "red"
      x: 25; y: 25; width: 50; height: 50
      rotation: 30
    }
}
```

代码的运行效果如图6-31所示。

Item还有一个transform属性，需要指定一个Transform元素的列表。Transform元素是一个基本类型，无法被直接实例化，可用的Transform类型有3个：Rotation、Scale和Translate，分别用来进行旋转、缩放和平移。这些元素可以通过专门的属性来进行更加高级的变换设置。其中，Rotation提供了坐标轴和原点属性，坐标轴有axis.x、axis.y和axis.z分别代表X轴、Y轴和Z轴，也就是说可以实现3D效果。原点由origin.x和origin.y来指定。简单的2D旋转是不需要指定坐标轴的，默认使用Z轴(axis{x: 0; y: 0; z: 0})即可。对于典型的3D旋转，既需要指定原点，也需要指定坐标轴。图6-32为原点和坐标轴的设置示意图。使用angle属性可以指定顺时针旋转的度数。下面的代码将一个蓝色矩形以Y轴为旋转轴进行了旋转。(项目源码路径：src\6\6-34\myRotation)

图6-31　项目旋转效果

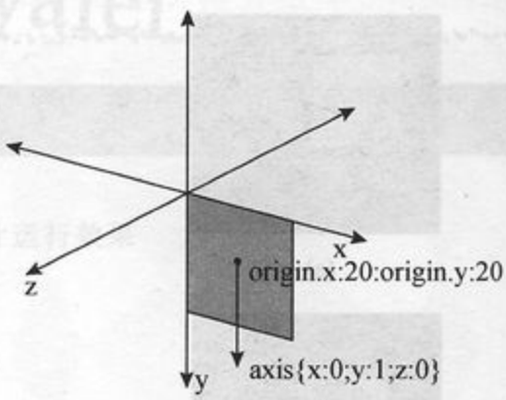



图6-32　旋转坐标轴示意图

```
import QtQuick 1.0

Rectangle {
  width: 100; height: 100
  color: "blue"
  transform: Rotation { origin.x: 30; origin.y: 30;
                        axis { x: 0; y: 1; z: 0 } angle: 72 }
}
```

代码的运行效果如图6-33所示。

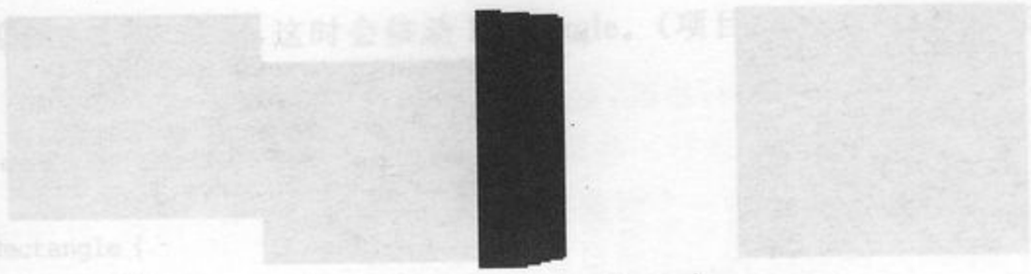


图 6-33　绕 Y 轴旋转效果

在 Scale 中提供了 origin.x 和 origin.y 属性来设置原点，另外还可以使用 xScale 和 yScale 来分别设置 X 轴和 Y 轴的比例因子，就是在 X 轴方向和 Y 轴方向的缩放值。例如下面的代码中将红色矩形在 X 轴方向放大了 3 倍。(项目源码路径：src\6\6-35\myScale)

```
import QtQuick 1.0

Rectangle {
    width: 100; height: 100
    color: "blue"
    Rectangle {
        width: 50; height: 50; x: 20; y: 20
        color: "red"
        transform: Scale { origin.x: 25; origin.y: 25; xScale: 3}
    }
}
```

代码运行效果如图 6-34 所示。而在 Translate 中提供了 x 和 y 属性来分别设置在 X 轴和 Y 轴方向的偏移量。在下面的例子中分别将两个矩形在 Y 轴方向进行了向上和向下的偏移。(项目源码路径：src\6\6-36\myTranslate)

```
import QtQuick 1.0

Row {
    Rectangle {
        width: 100; height: 100
        color: "blue"
        transform: Translate { y: 20 }
    }
    Rectangle {
        width: 100; height: 100
        color: "red"
        transform: Translate { y: -20 }
    }
}
```

代码的运行效果如图 6-35 所示。

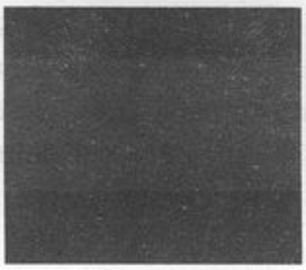


图6-34 沿X轴缩放效果

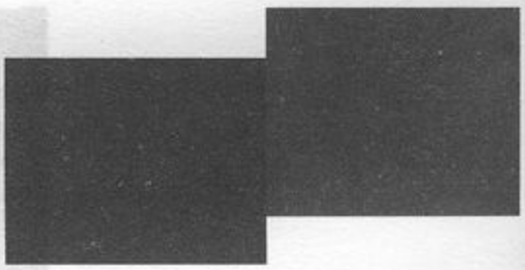


图6-35 沿Y轴平移效果

### 6.6.4 QML状态

用户界面用来显示不同场景中的界面，或者是改变它们的外观来响应用户的交互，通常情况下，有一系列变化是并发进行的。像这样的界面，可以看作是在其内部从一个状态改变到另一个状态。这普遍适用于各种界面元素，例如一个图片浏览器最初使用网格来显示多张图片，单击一张图片则进入到“详细”状态，这时会放大显示单张图片，而且用户界面也改变成了图片编辑界面。再比如，当一个按钮被按下时，便会改变到“按下”状态，按钮的颜色和位置会发生变化来产生一个被按下的外观。

在QML中，任何对象都可以在不同的状态间变化，每一个状态中都可以修改相关项目的属性来显示一个不同的配置，例如：

- 显示一些UI元素以及隐藏一些UI元素；
- 为用户呈现不同的动作；
- 开始、停止或者暂停动画；
- 执行一些需要在新的状态中使用的脚本；
- 为一个特定的项目改变一个属性；
- 显示一个不同的视图或者“场景”。

在不同的状态间改变时还可以使用切换(transitions)来实现动画效果，这个会在下一节讲解。所有基于Item的对象都有一个默认状态，还可以使用项目的states属性通过添加新的State对象来为其指定附加状态。每一个状态都有一个在本项目中唯一的名称，默认状态的状态名称为空字符串。要改变一个项目的当前状态，可以将state属性设置为要改变到的状态的名称。对于不是Item派生的对象可以通过StateGroup元素来使用状态。对应本节的内容，可以在Qt帮助中查看QML States关键字。

#### 1. 创建状态

要创建一个状态，可以向项目的states属性添加一个State对象，states属性包含了该项目状态的列表。例如，在下面的例子中Rectangle初始时被放在默认的(0, 0)的位置，这里定义了一个附加的moved状态，在这个状态中使用了PropertyChanges对象将矩形的位置更改为了(50, 50)。在MouseArea中进行单击就可以将状

态改变到 moved 状态，这时会移动 Rectangle。（项目源码路径：src\6\6 - 37\myState）

```
import QtQuick 1.0

Rectangle {
  id: myRect
  width: 200; height: 200
  color: "red"

  MouseArea {
    anchors.fill: parent
    onClicked: myRect.state = "moved"
  }

  states: [
    State {
      name: "moved"
      PropertyChanges { target: myRect; x: 50; y: 50 }
    }
  ]
}
```

State 项目定义了在新状态中所要进行的所有改变，可以指定被改变的附加属性，或者为其他对象创建附加的 PropertyChanges，也可以修改其他对象的属性，而不仅仅是拥有状态的对象。例如：

```
Rectangle {
  ...
  states: [
    State {
      name: "moved"
      PropertyChanges { target: myRect; x: 50; y: 50; color: "blue" }
      PropertyChanges { target: someOtherItem; width: 1000 }
    }
  ]
}
```

就像在前面提到过的，为了方便，如果一个项目只有一个状态，那么可以省略方括号：

```
Item {
  ...
  states: State {
    ...
  }
```

```
}
```

State不仅限于对属性值进行修改，它还可以：

- 使用StateChangeScript运行一些脚本；
- 使用PropertyChanges为一个对象重写现有的信号处理器；
- 使用PropertyChanges为一个项目重定义父项目；
- 使用AnchorChanges修改锚的值。

在Qt提供的States and Transitions example示例程序中演示了怎样声明一组基本的状态并在它们之间使用动画进行切换，可以参考一下。

**2. 默认状态**

在前面的例子中其实也可以简单地在MouseArea的onClicked处理器中将Rectangle的位置设置为(50, 50)。然而，除了进行批量的属性变化，QML的另一个特色就是可以使项目恢复到默认状态。在默认状态中包含了项目所有的初始化属性值。例如，在下面的代码中当鼠标被按下时Rectangle移动到(50, 50)点，而当鼠标释放时重新移回原来的位置，这个可以使用when属性来实现。

```
Rectangle {
  ...
  MouseArea {
    id: mouseArea
    anchors.fill: parent
  }

  states: State {
    name: "moved"; when: mouseArea.pressed
    ...
  }
}
```

当when属性设置的表达式值为true时，项目会设置为该状态。另外，一个项目可以为其state属性指定一个空字符串来明确地将其状态设置为默认状态。例如，如果不使用when属性，前面的代码也可以写为：

```
Rectangle {
  ...
  MouseArea {
    anchors.fill: parent
    onPressed: myRect.state = "moved";
    onReleased: myRect.state = "";
  }

  states: State {
```

```
        name: "moved"
        ...
    }
}
```

很明显，使用 when 属性比使用信号处理器来分配状态更加简单，更符合声明式的语言。所以建议在这种情况下使用前面 when 属性的方法。

### 6.6.5　QML 动画

在 QML 中，可以在对象的属性值上应用动画对象来随着时间逐渐改变它们从而创建动画。动画对象可以从一组内建的动画元素进行创建，可以用来为多种类型的属性值产生动画。此外，动画对象可以通过不同的方式进行应用，这依赖于它们所需要的背景。对应本节的内容，可以在 Qt 帮助中查看 QML Animation 关键字。

**1. 动画类型**

一个动画创建的方式取决于它所需要的背景。假设一个 Rectangle 的运动，就是改变它的 x 或者 y 属性的值，动画的语义的不同依赖于是否要创建以下几点：

- Rectangle 一旦创建该动画就要将其移动到一个已知的位置；
- 动画只有在 Rectangle 被外部源移动时才会触发，例如，当单击鼠标时，产生动画移动到鼠标位置；
- 只有在接收到一个特定的信号后才触发该动画；
- 作为一个独立的动画，虽然没有绑定 Rectangle 的运动，但是可以从脚本中进行开启和停止；
- 只有在状态改变时才会触发该动画。

为了支持这些不同类型的动画方式，QML 提供了多种方式来定义一个动画：

- 使用属性值源来创键一个动画，可以立即为一个指定的属性使用动画；
- 使用行为动画，当一个属性改变值时触发；
- 在一个信号处理器中创建，当接收到一个信号时触发；
- 作为一个独立动画，可以在脚本中进行开始/停止，也可以重新绑定到不同的对象；
- 使用切换，在不同状态间提供动画。

下面分别介绍这些方式。在这里都会使用到 PropertyChanges 元素，它可以用来创建一个动画，后面会对其详细介绍。

**2. 动画作为属性值的来源**

一个动画被应用为属性值源(property value source)，要使用“动画 on 属性”语法。例如，下面的例子中 Rectangle 的运动就使用了这个方法。(项目源码路径：src\6\6-38\myAnimation)

```
import QtQuick 1.0

Rectangle {
  width: 100; height: 100
  color: "red"
  PropertyAnimation on x { to: 50; duration: 1000; loops: Animation.Infinite }
  PropertyAnimation on y { to: 50; duration: 1000; loops: Animation.Infinite }
}
```

这里在Rectangle的x和y属性上应用了PropertyAnimation来使它们从当前值(0)在1000毫秒中使用动画变化到50。Rectangle一旦加载完成就会开启该动画。要指定一个自定义的值而不是使用当前值作为起始值,可以设置PropertyAnimation的from属性。这里的loops属性指定为Animation.Infinite,表明该动画是无限循环的。指定一个动画作为属性值源,在一个对象加载完成后立即就对一个属性使用动画变化到一个指定的值的情况是非常有用的。

**3. 行为动画**

经常在一个特定的属性值改变时要应用一个动画,在这种情况下,可以使用一个Behavior为一个属性改变指定一个默认的动画。下面来看一个例子。(项目源码路径:src\6\6-39\myAnimation)

```
import QtQuick 1.0

Item {
  width: 100; height: 100
  Rectangle {
    id: rect
    width: 100; height: 100
    color: "red"

    Behavior on x { PropertyAnimation { duration: 500 } }
    Behavior on y { PropertyAnimation { duration: 500 } }
  }

  MouseArea {
    anchors.fill: parent
    onClicked: { rect.x = mouse.x; rect.y = mouse.y }
  }
}
```

这里Rectangle拥有一个Behavior对象应用到了它的x和y属性。每当这些属性改变(这里是在父项目Item中单击时),在Behavior中的PropertyAnimation对象就会应用到这些属性上,从而使Rectangle使用动画效果移动到一个新的位置。行为动画是在每次响应一个值的变化时触发的。对这些属性的任何改变都会触发它们

的动画，如果 x 或 y 绑定到了其他属性上，这些属性改变时也会触发动画。在一些情况下还可以通过设置 enabled 属性来停用 Behavior。注意，在这里 PropertyAnimation 的 from 和 to 属性是不需要定义的，因为这些值已经提供了，分别是 Rectangle 的当前值和 onClicked 处理器中设置的新值。关于行为动画的使用，也可以参考一下 Qt 提供的 Behaviors example 示例程序。

**4. 在信号处理器中的动画**

可以在一个信号处理器中创建一个动画，并在接收到信号时触发。下面来看一个例子。（项目源码路径：src\6\6 - 40\myAnimation）

```
import QtQuick 1.0

Rectangle {
  id: rect
  width: 100; height: 100
  color: "red"

  MouseArea {
    anchors.fill: parent
    onClicked: PropertyAnimation { target: rect; properties: "x,y";
                                   to: 50; duration: 1000 }
  }
}
```

当 MouseArea 被单击时则触发 PropertyAnimation，在 1000 毫秒内使用动画将 x 和 y 的属性改变为 50。因为动画没有绑定到一个特定的对象或者属性，所以必须指定 target 和 property（或者 targets 和 properties）属性的值。而且还需要使用 to 属性来指定新的 x 和 y 的值。

**5. 独立动画**

动画也可以像一个普通的 QML 对象一样进行创建，而不需要绑定到任何特定的对象和属性。下面的例子使用了一个 PropertyAnimation 对象，当 Rectangle 被单击时会启动动画。（项目源码路径：src\6\6 - 41\myAnimation）

```
import QtQuick 1.0

Rectangle {
  id: rect
  width: 100; height: 100
  color: "red"

  PropertyAnimation {
    id: animation
    target: rect
```

```
        properties: "x,y"
        duration: 1000
    }

    MouseArea {
        anchors.fill: parent
        onClicked: {
            animation.to = 50;
            animation.running = true;
        }
    }
}
```

一个独立的动画对象默认是没有运行的，必须使用 running 属性或者 start()和 stop()函数来明确地运行它。因为动画没有绑定到一个特殊的对象或属性上，所以必须定义 target 和 property(或者 targets 和 properties)属性的值。也需要使用 to 属性来指定新的 x 和 y 的值。独立动画在不是对一个单一对象属性进行动画而且动画需要明确开始和停止的情况下是非常有用的。

### 6. 切　换

切换用来设置当状态改变时的动画，要创建一个切换，需要定义一个 Transition 对象，然后将其添加到项目的 transitions 属性。下面来看一个例子。(项目源码路径：src\6\6-42\myAnimation)

```
import QtQuick 1.0

Rectangle {
    id: rect
    width: 100; height: 100
    color: "red"

    MouseArea {
        anchors.fill: parent
        onClicked: rect.state = "moved"
    }

    states: State {
        name: "moved"
        PropertyChanges { target: rect; x: 50; y: 50 }
    }

    transitions: Transition {
        PropertyAnimation { properties: "x,y"; duration: 1000 }
    }
```

```
}
```

在 moved 状态中的 PropertyChanges 对象定义了当 Rectangle 在该状态时其位置应该改变为(50，50)。当 Rectangle 改变到 moved 状态时，Transition 将被触发，切换的 PropertyAnimation 将会使用动画将 x 和 y 属性改变到它们的新值。注意这里并没有为 PropertyAnimation 设置任何 from 和 to 属性的值，在状态改变的开始之前和结束之后会自动设置这些属性为 x 和 y 的值。另外，PropertyAnimation 并不需要指定 target 对象，这样任何对象的 x 或者 y 的值在状态改变时进行更改都会使用动画。不过，也可以指定一个 target 来为特定的对象使用动画。在 Transition 中的顶级动画会并行运行，要想一个接一个地运行，可以使用 SequentialAnimation，这个会在后面的内容中讲到。

**7. 动画元素**

需要在内建的动画元素中选择一个来创建一个动画，虽然在前面的例子中都使用了 PropertyAnimation 进行演示，不过也可以根据属性的类型以及是否需要一个或者多个动画来选择使用其他的元素。所有的动画元素都继承自 Animation 元素，尽管无法直接创建 Animation 对象，不过它为动画元素提供了必要的属性和函数。例如，它允许使用 running 属性和 start()和 stop()函数来控制动画的开始和停止，也可以通过 loops 属性定义动画的循环次数。

**8. 属性动画元素**

PropertyAnimation 是用来为属性提供动画的最基本的动画元素，可以用来为 real、int、color、rect、point、size 和 vector3d 等属性设置动画，被 NumberAnimation、colorAnimation、RotationAnimation 和 Vector3dAnimation 等元素继承。NumberAnimation 对 real 和 int 属性提供了更高效的实现；Vector3dAnimation 对 vector3d 属性提供了更高效的支持；而 ColorAnimation 和 RotationAnimation 分别对 color 和 rotation 属性变化动画提供了特定的属性支持。例如，ColorAnimation 允许颜色值设置 from 和 to 属性。(项目源码路径：src\6\6 - 43\myAnimation)

```
import QtQuick 1.0

Rectangle {
  width: 100; height: 100

  ColorAnimation on color { from: "red"; to: "yellow"; duration: 1000 }
}
```

RotationAnimation 允许设定旋转的方向。(项目源码路径：src\6\6 - 44\myAnimation)

```
import QtQuick 1.0
```

```
Item {
    width: 300; height: 300

    Rectangle {
        width: 100; height: 100; anchors.centerIn: parent
        color: "red"

        RotationAnimation on rotation { to: 90;
            direction: RotationAnimation.Clockwise }
    }
}
```

另外，还有下面几种专门的动画元素可以使用：

- SmoothedAnimation：它是一个专门的 NumberAnimation，当目标值改变时在动画中为其提供了一个平滑的变化；
- SpringAnimation：提供了一个像弹簧一样的动画，可以设置 mass、damping 和 epsilon 等属性；
- ParentAnimation：用来在改变父项目时产生动画（对应 ParentChange 元素）；
- AnchorAnimation：用来在改变锚时产生动画（对应 AnchorChanges 元素）。

关于这些动画元素的使用，可以查看其帮助文档。对于任何基于 PropertyAnimation 的动画都可以通过设置 easing 属性来控制在属性值动画中使用的缓和曲线。它们可以影响这些属性值的动画效果，提供一些如反弹、加速和减速等视觉效果。例如，在下面的例子中通过使用 Easing.OutBounce 来创建了一个动画到达目标值时的反弹效果。（项目源码路径：src\6\6-45\myAnimation）

```
import QtQuick 1.0

Rectangle {
    width: 100; height: 100
    color: "red"
    PropertyAnimation on x { to: 50; duration: 1000;
        easing.type: Easing.OutBounce }
    PropertyAnimation on y { to: 50; duration: 1000;
        easing.type: Easing.OutBounce }
}
```

Qt 提供的 easing example 示例程序演示了所有缓和类型的效果，可以参考。

**9. 组合动画**

多个动画可以组合成一个单一的动画，这可以使用 ParallelAnimation 或者 SequentialAnimation 动画组元素中的一个来实现。在 ParallelAnimation 中的动画会同时进行，而在 SequentialAnimation 中的动画会一个接一个地运行。要想运行多个动画，可以在一个动画组中定义。例如，在下面的例子中创建了 SequentialAnima-

tion 来一个接一个地运行 3 个动画：NumberAnimation、PauseAnimation 和 NumberAnimation。这里 SequentialAnimation 是作为属性值源动画应用在了 image 的 y 属性上，所以动画会在图片加载完成后就立即执行。（项目源码路径：src\6\6 - 46\myAnimation）

```
import QtQuick 1.0

Rectangle {
    id: rect
    width: 120; height: 200

    Image {
        id: img
        source: "qt.png"
        anchors.horizontalCenter: parent.horizontalCenter
        y: 0

        SequentialAnimation on y {
            loops: Animation.Infinite
            NumberAnimation { to: rect.height - img.height;
                easing.type: Easing.OutBounce; duration: 2000 }
            PauseAnimation { duration: 1000 }
            NumberAnimation { to: 0; easing.type: Easing.OutQuad;
                duration: 1000 }
        }
    }
}
```

因为 SequentialAnimation 是应用在 y 属性上的，所以在组中的独立的动画也会自动应用在 y 属性上。动画组还可以嵌套，例如，下面的例子是一个相对复杂的动画，它同时使用了顺序和并行动画。（项目源码路径：src\6\6 - 47\myAnimation）

```
import QtQuick 1.0

Rectangle {
    id: redRect
    width: 100; height: 100
    color: "red"

    MouseArea { id: mouseArea; anchors.fill: parent }

    states: State {
        name: "pressed"; when: mouseArea.pressed
        PropertyChanges { target: redRect; color: "blue";
```

```
        y: mouseArea.mouseY; width: mouseArea.mouseX }
    }
    transitions: Transition {

      SequentialAnimation {
        ColorAnimation { duration: 200 }
        PauseAnimation { duration: 100 }

        arallelAnimation {
          NumberAnimation {
            duration: 500
            easing.type: Easing.OutBounce
            targets: redRect
            properties: "y"
          }

          NumberAnimation {
            duration: 800
            easing.type: Easing.InOutQuad
            targets: redRect
            properties: "width"
          }
        }
      }
    }
  }
```

一旦独立的动画被放入 SequentialAnimation 或者 ParallelAnimation，那么它们就不能再独立开启或者停止。顺序或者并行动画必须作为一个组进行开启和停止。可以参考 Qt 自带的 Animation basics example 示例程序。

**10. 其他动画元素**

另外，QML 还为动画提供了其他一些有用的元素：

- PauseAnimation：在动画中间进行暂停；
- ScriptAction：允许在动画中执行 JavaScript，也可以和 StateChangeScript 一起使用来重用已经存在的脚本；
- PropertyAction：在动画中间立即改变一个属性而不对该属性的变化使用动画。

可以在这些元素的帮助文档中查看详细介绍。

## 6.6.6　Flickable 和 Flipable

### 1. 弹动效果 Flickable

QML 中提供了一个 Flickable 元素，它可以将其子项目设置在一个可以拖拽和弹动的界面上，使得子项目的视图可以滚动。这种行为构成了被设计用来显示大量子项目元素的基础，例如 ListView 和 GridView。在传统的用户界面中，视图可以使用滚动条和箭头按钮等标准控件来滚动。在某些情况下，也可以按下鼠标按钮的同时移动光标来拖动视图。但是在基于触摸的用户界面中，拖拽动作经常使用弹动动作来实现，当用户已经停止触摸视图后它还会继续滚动。Flickable 不会自动裁剪它的内容，如果不是将它用作全屏项目，可以将 clip 属性设置为 true 来隐藏超出区域的内容。下面例子使用了一个很小的视图来显示一个很大的图片，这时可以通过拖拽和弹动图片来显示其他的部分。（项目源码路径：src\6\6 - 48\myFlickable）

```
import QtQuick 1.0

Flickable {
    width: 200; height: 200
    contentWidth: image.width; contentHeight: image.height

    Image { id: image; source: "bigImage.jpg" }
}
```

这里的 contentWidth 和 contentHeight 属性用来设置可以进行拖拽的内容的大小，这个一般设置为要在 Flickable 中显示内容的整体大小。在下面的例子中，只在一个区域中显示 Flickable 元素，这时为了不让其在指定区域以外显示，将其 clip 属性设置为 true。（项目源码路径：src\6\6 - 49\myFlickable）

```
import QtQuick 1.0

Rectangle {
    width: 360
    height: 360
    color: "blue"

    Flickable {
        width: 200; height: 200
        contentWidth: image.width; contentHeight: image.height
        clip: true

        Image { id: image; source: "bigImage.jpg" }
    }
}
```

代码的运行效果如图6-36所示。

有时希望在Flickable中显示一个滚动条，这可以通过visibleArea属性来实现，这个属性分为visibleArea.xPosition、visibleArea.widthRatio、visibleArea.yPosition和visibleArea.heightRatio。它们都是只读属性，描述了当前可视区域的位置和大小。大小被定义为当前可视窗口占整个视图的百分比，从0.0到1.0。页面位置一般是从0.0到1.0减去大小比，例如yPosition的范围是0.0～1.0－heightRatio。然而，内容可以拖拽到正常的范围之外，所以页面位置也可能在正常范围之外。下面的例子中使用了这个属性来实现了一个垂直方向的黑色滚动条。(项目源码路径：src\6\6-50\myFlickable)

```
import QtQuick 1.0

Rectangle {
    width: 300; height: 300

    Flickable {
        id: flickable
        width: 300; height: 300
        contentWidth: image.width; contentHeight: image.height
        clip: true

        Image { id: image; source: "bigImage.jpg" }
    }

    Rectangle {
        id: scrollbar
        anchors.right: flickable.right
        y: flickable.visibleArea.yPosition * flickable.height
        width: 10
        height: flickable.visibleArea.heightRatio * flickable.height
        color: "black"
    }
}
```

代码的运行效果如图6-37所示。对应这个例子，也可以参考一下Qt自带的scrollbar example示例程序。另外，使用flickDeceleration可以设置弹动的速度，使用flickableDirection可以设置弹动的方向等。对于Flickable元素其他的属性和信号的使用，可以参考其帮助文档。

图 6－36　Flickable 的 clip 效果

图 6－37　在 Flickable 中添加滚动条

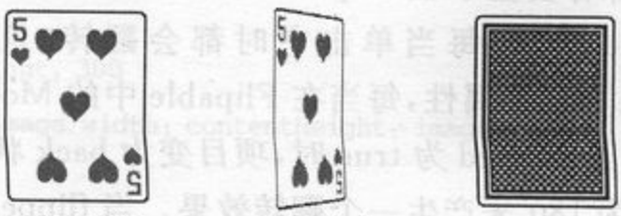
**2. 翻转效果 Flipable**

Flipable 是一个可以明显在其正面和反面之间进行翻转的项目，就像一张卡片。这是通过同时使用 Rotation、State 和 Transition 等元素来产生翻转效果的。front 和 back 属性分别用来保存要显示在 Flipable 项目正面和反面的项目。下面的例子中显示了一个 Flipable 项目，每当单击它时都会翻转，且围绕 y 轴进行旋转。Flipable 有一个 flipped 布尔值属性，每当在 Flipable 中的 MouseArea 上单击鼠标时都会切换该属性的值。当 flipped 为 true 时，项目变为 back 状态，在这个状态，Rotation 的 angle 属性改变为 180°来产生一个翻转效果。当 flipped 为 false 时，项目恢复到默认状态，这时 angle 的值为 0。（项目源码路径：src\6\6－51\myFlipable）

```
import QtQuick 1.0

Flipable {
    id: flipable
    width: 240
    height: 240

    property bool flipped: false

    front: Image { source: "front.png"; anchors.centerIn: parent }
    back: Image { source: "back.png"; anchors.centerIn: parent }

    transform: Rotation {
        id: rotation
        origin.x: flipable.width/2
        origin.y: flipable.height/2
        axis.x: 0; axis.y: 1; axis.z: 0
        angle: 0
    }
```

```
    states: State {
        name: "back"
        PropertyChanges { target: rotation; angle: 180 }
        when: flipable.flipped
    }

    transitions: Transition {
        NumberAnimation { target: rotation; property: "angle"; duration: 4000 }
    }

    MouseArea {
        anchors.fill: parent
        onClicked: flipable.flipped = ! flipable.flipped
    }
}
```

代码的运行效果如图 6-38 所示。对应这个例子，也可以参考一下 Qt 提供的 Flipable example 示例程序。



(a)翻转前　(b)翻转中　(c)翻转后

图 6-38　Flipable 运行效果

## 6.7　QML 中的模型/视图

在《Qt Creator 快速入门》的第 16 章讲述了 Qt 中用来存储和显示数据的模型/视图编程。在 QML 中对于数据的存储和显示也同样使用了模型/视图框架，其基本构成与 C++中的是一样的。所以在学习本节以前，也可以先学习一下《Qt Creator 快速入门》中第 16 章的内容，从而加强对模型/视图概念的理解。

### 6.7.1　QML 数据模型

QML 中的一些视图项目(如 ListView、GridView 和 Repeater 等)需要使用数据模型来为其提供数据进行显示。这些项目通常也需要一个委托(delegate)组件来为模型中的每一个条目创建一个实例。模型可以是静态的，也可以进行动态的修改、插入、移除或者移动项目。对应本节的内容，可以在 Qt 帮助中参考 QML Data Models 关键字。

通过命名委托绑定的数据角色来为委托提供数据。例如，在下面的例子中 ListModel 拥有两个角色，type 和 age。还有一个 ListView 中包含了一个委托，委托绑定了这些角色来显示它们的值。（项目源码路径：src\6\6-52\myDataModel）

```
import QtQuick 1.0

Item {
  width: 200; height: 250

  ListModel {
    id: myModel
    ListElement { type: "Dog"; age: 8 }
    ListElement { type: "Cat"; age: 5 }
  }

  Component {
    id: myDelegate
    Text { text: type + ", " + age }
  }

  ListView {
    anchors.fill: parent
    model: myModel
    delegate: myDelegate
  }
}
```

这里 ListView 用来进行显示，在其中的数据模型 model 用来提供数据，委托 delegate 用来设置数据的显示方式，这里分别指定为了 myModel 和 myDelegate 对象。在 ListModel 中，可以使用 ListElement 添加条目，每一个条目中可以有多种类型的角色，比如这里有两个：type 和 age，并且分别指定了它们的值。而委托可以使用一个组件来实现，在委托中可以直接绑定数据模型中的角色，比如这里将 type 和 age 的值显示在了一个 Text 文本中。委托的概念可以参考《Qt Creator 快速入门》中第 16 章的内容，简单来说，就是数据模型中每一个条目显示的时候都会使用委托提供的显示方式进行显示，可以看作一个模板。

如果模型的属性和委托的属性出现了名字冲突，那么角色可以通过限定模型名称来访问。例如，如果前面例子中委托中的 Text 元素也有一个 type 或者 age 属性，那么其文本将会显示为它的属性值，而不是模型中 type 和 age 的值。在这种情况下，可以使用 model.type 和 model.age 来确保委托中可以显示正确的模型中的值。

在委托中还可以使用一个特殊的 index 角色，它包含了模型中条目的索引值。注意，如果条目已经从模型中移除，那么其索引值为－1。所以如果在委托中绑定了 index 角色，那么一定要注意它的值有可能变为－1 的情况。

如果模型中没有包含命名的角色，那么可以通过 modelData 角色来提供数据。对于只有一个角色的模型也可以使用 modelData。这种情况下 modelData 角色与命名角色包含了相同的数据。在 QML 的内建元素中提供了多个类型的数据模型，另外，还可以使用 C++定义模型，然后在 QML 组件中使用。

**1. ListModel**

ListModel 是一个简单的具有层次的元素，可以使用 ListElement 属性来指定可用的角色。例如：

```
ListModel {
    id: fruitModel

    ListElement {
        name: "Apple"
        cost: 2.45
    }
    ListElement {
        name: "Orange"
        cost: 3.25
    }
    ListElement {
        name: "Banana"
        cost: 1.95
    }
}
```

这里的模型有两个角色：name 和 cost。它们可以绑定到 ListView 的委托上，例如：

```
ListView {
    anchors.fill: parent
    model: fruitModel
    delegate: Row {
        Text { text: "Fruit: " + name }
        Text { text: "Cost: $" + cost }
    }
}
```

ListModel 提供了函数来直接使用 JavaScript 操纵 ListModel。在这种情况下，第一个插入的条目决定了使用该模型的视图中可用的角色。例如，如果创建了一个空的 ListModel，然后使用 JavaScript 进行了填充，那么第一个插入的条目中包含的角色就是将来在视图中显示的角色：

```
ListModel { id: fruitModel }
...
```

```
MouseArea {
  anchors.fill: parent
  onClicked: fruitModel.append({"cost": 5.95, "name":"Pizza"})
}
```

当 MouseArea 被单击时，fruitModel 将会有两个角色：cost 和 name。即便以后再有其他的角色加入，在视图中也只会处理前两个角色。如果要重置在模型中可用的角色，可以调用 ListModel::clear()。

**2. XmlListModel**

XmlListModel 允许从一个 XML 数据源创建一个模型，可以通过 XmlRole 元素来指定一个角色。该数据模型一般用来显示从网络上获取的数据，因为网络上很多数据都可以使用 XML 格式进行读取，这样就可以使用 XmlListModel 对其进行解析，然后显示出自己想要的数据。所以使用该模型可以非常容易创建基于网络的应用程序，这也得益于 QML 中透明的网络传输以及内部的多线程支持。例如，下面的例子中将网络上的数据进行了显示，模型中包含了 title、link 和 description 这 3 个角色。（项目源码路径：src\6\6 - 53\myDataModel）

```
import QtQuick 1.0

Rectangle {
  width: 360
  height: 360

  XmlListModel {
    id: feedModel
    source: "http://rss.news.yahoo.com/rss/oceania"
    query: "/rss/channel/item"
    XmlRole { name: "title"; query: "title/string()" }
    XmlRole { name: "link"; query: "link/string()" }
    XmlRole { name: "description"; query: "description/string()" }
  }

  ListView {
    anchors.fill: parent
    model: feedModel
    delegate: Column {
      Text { text: "title: " + title }
      Text { text: "link: $" + link }
      Text { text: "description: $" + description }
    }
  }
}
```

XmlListModel用来为XML数据创建一个只读的模型，可以用作视图元素(例如ListView、PathView，GridView)的数据源，也可以用于其他与模型数据交互的元素(例如Repeater)。下面来看一下它是怎样解析XML数据的。假设在http://www.mysite.com/feed.xml上有一个XML文档，其内容如下：

```
<? xml version = "1.0" encoding = "utf - 8"? >
<rss version = "2.0">
  ...
  <channel>
    <item>
      <title>A blog post</title>
      <pubDate>Sat, 07 Sep 2010 10:00:01 GMT</pubDate>
    </item>
    <item>
      <title>Another blog post</title>
      <pubDate>Sat, 07 Sep 2010 15:35:01 GMT</pubDate>
    </item>
  </channel>
</rss>
```

下面从该数据中使用XmlListModel来创建一个模型：

```
import QtQuick 1.0

XmlListModel {
  id: xmlModel
  source: "http://www.mysite.com/feed.xml"
  query: "/rss/channel/item"

  XmlRole { name: "title"; query: "title/string()" }
  XmlRole { name: "pubDate"; query: "pubDate/string()" }
}
```

这里query的值为“/rss/channel/item”，其指定了这个XmlListModel应该从该XML文档的每一个<item>创建一个模型条目。而XmlRole对象定义了模型条目的属性。这里每一个条目都包含了title和pubDate属性，它们与相应的<item>中的title和pubDate值对应。这里还要指定值的类型，比如title是字符串，则使用string()，如果是数值，那么使用number()。如果要查询一个元素的属性，比如<book type="Paperback">中的type属性，那么可以使用query："@type/string()"进行查询，也就是在type前加上一个@标志。如果想了解更多的查询的相关知识，则需要查阅XPath表达式的相关知识。注意XmlListModel数据是异步进行加载的，当加载完成时它的status属性会更改为XmlListModel.Ready。这也就意味着当在视图中使用XmlListModel时，只有等到模型被加载完成后视图才会被填充。

还可以将特定的角色定义为 keys，这样当调用 reload()重新加载数据时，模型只会在这些角色包含新的值时才进行添加或者刷新。例如：

```
XmlRole { name: "pubDate"; query: "pubDate/string()"; isKey: true }
```

当调用 reload()时，模型只会添加和重新加载模型中那些“pubDate”值没有显示的条目。这对于显示增量更新内容的 XML 文档(例如 RSS 订阅)是十分有用的，可以避免在视图中重绘模型中的全部内容。可以参考 Qt 提供的 RSS News 演示程序。

**3. VisualItemModel**

VisualItemModel 允许使用 QML 项目作为模型。这个模型同时包含了数据和委托，在 VisualItemModel 中的子项目提供了委托的内容，该模型没有提供任何角色。例如：

```
VisualItemModel {
    id: itemModel
    Rectangle { height: 30; width: 80; color: "red" }
    Rectangle { height: 30; width: 80; color: "green" }
    Rectangle { height: 30; width: 80; color: "blue" }
}

ListView {
    anchors.fill: parent
    model: itemModel
}
```

另外，QML 中还可以使用 C++代码定义的模型，可以在 Qt 帮助文档中参考 QML Data Models 关键字对应的文档的 C++ Data Models 部分的内容。

**4. 在委托中访问视图和模型**

可以在委托中访问使用该委托的视图及其属性，例如使用 ListView 时，可以在委托中使用 ListView. view 来进行访问。而且，还可以在委托中访问模型及其属性，例如，使用 ListView 时，可以使用 ListView. view. model 来访问。这对于在多个视图中使用相同的委托但对于每一个视图又想有不同特色的情况是非常有用的。例如，在下面的例子中，委托显示了模型的 language 属性，而且使用了视图的 fruit_color 属性来设置颜色。(项目源码路径：src\6\6 - 54\myDataModel)

```
import QtQuick 1.0

Rectangle {
    width: 200; height: 200

    ListModel {
        id: fruitModel
```

```
    property string language: "en"
    ListElement {
      name: "Apple"
      cost: 2.45
    }
    ListElement {
      name: "Orange"
      cost: 3.25
    }
    ListElement {
      name: "Banana"
      cost: 1.95
    }
  }

  Component {
    id: fruitDelegate
    Row {
      Text { text: " Fruit: " + name; color: ListView.view.fruit_color }
      Text { text: " Cost: $" + cost }
      Text { text: " Language: " + ListView.view.model.language }
    }
  }

  ListView {
    property color fruit_color: "green"
    model: fruitModel
    delegate: fruitDelegate
    anchors.fill: parent
  }
}
```

另一种情况是，在委托中的一些动作(例如点击鼠标)需要更新模型中的数据，这时可以在模型中定义一个函数，例如：

```
setData(int row, const QString & field_name, QVariant new_value)
```

然后在委托中合适的位置调用该函数：

```
ListView.view.model.setData(index, field, value)
```

这里 field 需要指定为要更新的域的名字，而 value 是新的值。

### 6.7.2 在 QML 中呈现数据

Qt Quick 包含了一组可以使用不同方式呈现数据的标准项目。对于简单的用

户界面，可以将 Repeaters 和 Positioners 一起使用来包含一些数据并将它们排列在用户界面上。然而，当涉及大量的数据时，更好的办法是使用模型和视图来进行显示。

视图是一个包含条目集合的可滚动容器，功能丰富，支持典型应用程序中的多种使用情况，而且还可以进行自定义风格和行为来满足需求。在 Qt Quick 基本图形元素中提供了一组标准的视图：

- ListView 在水平或者垂直列表中排列条目；
- GridView 在可用空间中将条目排列在一个网格中；
- PathView 在路径上排列条目。

与这些视图不同，WebView 不是一个功能完整的视图项目，需要和 Flickable 项目一起使用来创建一个视图，实现的功能与网页浏览器相似。对应本节的内容，可以在 Qt 帮助中查看 Presenting Data with QML 关键字。

**1. ListView**

ListView 可以显示一个水平或者垂直放置条目的条目列表。ListView 拥有一个模型 model 属性，用来定义要显示的数据，还有一个委托 delegate 属性，用来定义数据显示的方式。ListView 默认具有弹动效果，因为它继承自 Flickable 项目。上一节已经多次使用过 ListView 元素了，那些例子中模型都是和视图定义在同一个文件的，其实模型也可以定义在一个独立的文件中，然后像一般的组件一样来使用它。下面来看一个例子。（项目源码路径：src\6\6 - 55\myListView）

```
import QtQuick 1.0

Rectangle {
  width: 180; height: 200

  Component {
    id: contactDelegate
    Item {
      width: 180; height: 40
      Column {
        Text { text: '<b>Name:</b> ' + name }
        Text { text: '<b>Number:</b> ' + number }
      }
    }
  }
  ListView {
    anchors.fill: parent
    model: ContactModel {}
    delegate: contactDelegate
```

```
        highlight: Rectangle { color: "lightsteelblue"; radius: 5 }
        focus: true
    }
}
```

下面向项目中添加新的QML文件ContactModel.qml,并将其内容更改如下:

```
import QtQuick 1.0

ListModel {
    ListElement {
        name: "Bill Smith"
        number: "555 364"
    }
    ListElement {
        name: "John Brown"
        number: "555 846"
    }
    ListElement {
        name: "Sam Wise"
        number: "555 0473"
    }
}
```

在前面的代码中ListView创建了一个ContactModel组件作为其模型。使用highlight属性可以指定当前选择项目的高亮显示,这里使用了蓝色的Rectangle。而将focus属性设置为true是为了确保可以使用键盘来导航列表视图。代码的运行效果如图6-39所示。

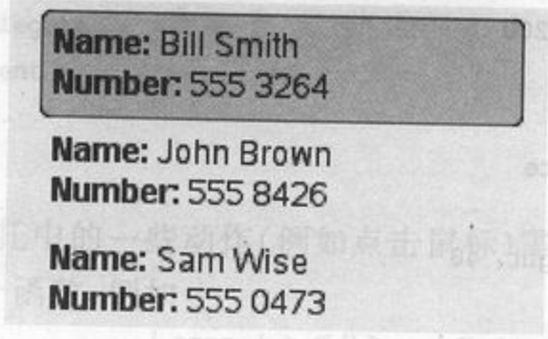



图6-39 ListView高亮效果

ListView在委托的根项目中附加了多个属性,例如ListView.isCurrentItem可以用来判断一个条目是否是当前条目。在下面的例子中可以发现根委托项目可以使用ListView.isCurrentItem来直接访问该属性,而contactInfo对象必须使用wrapper.ListView.isCurrentItem来访问该属性。

```
ListView {
    width: 180; height: 200
```

```
    Component {
        id: contactsDelegate
        Rectangle {
            id: wrapper
            width: 180
            height: contactInfo.height
            color: ListView.isCurrentItem ? "black" : "red"
            Text {
                id: contactInfo
                text: name + ": " + number
                color: wrapper.ListView.isCurrentItem ? "red" : "black"
            }
        }
    }

    model: ContactModel {}
    delegate: contactsDelegate
    focus: true
}
```

代码的运行效果如图 6-40 所示。在视图中是不会自动裁剪的，如果不想让视图的内容在拖拽时显示到视图以外的区域，那么可以将 clip 属性设置为 true。另外，ListView 还提供了 decrementCurrentIndex()和 incrementCurrentIndex()函数来进行条目的自动滚动。关于 ListView 更多的内容，可以参考其帮助文档，也可以参考 Qt 的 ListView Example 示例程序。


图 6-40　当前条目处理效果

**2. GridView**

GridView 与 ListView 的概念以及使用方法是非常相似的，主要的不同之处就是 GridView 是将条目排列成网格。下面来看一个例子。(项目源码路径：src\6\6-56\myGridView)

```
import QtQuick 1.0

Rectangle {
    width: 300; height: 400
    color: "white"

    ListModel {
        id: appModel
```

```
        ListElement { name: "Music"; icon: "AudioPlayer_48.png" }
        ListElement { name: "Movies"; icon: "VideoPlayer_48.png" }
        ListElement { name: "Camera"; icon: "Camera_48.png" }
        ListElement { name: "Calendar"; icon: "DateBook_48.png" }
        ListElement { name: "Messaging"; icon: "EMail_48.png" }
        ListElement { name: "Todo List"; icon: "TodoList_48.png" }
        ListElement { name: "Contacts"; icon: "AddressBook_48.png" }
    }

    Component {
        id: appDelegate
        Item {
            width: 100; height: 100
            Image {
                id: myIcon
                y: 20; anchors.horizontalCenter: parent.horizontalCenter
                source: icon
            }
            Text {
                anchors { top: myIcon.bottom;
                    horizontalCenter: parent.horizontalCenter }
                text: name
            }
        }
    }

    Component {
        id: appHighlight
        Rectangle { width: 80; height: 80; color: "lightsteelblue" }
    }

    GridView {
        anchors.fill: parent
        cellWidth: 100; cellHeight: 100
        highlight: appHighlight
        focus: true
        model: appModel
        delegate: appDelegate
    }
}
```

代码的运行效果如图6-41所示。关于GridView更多的内容，可以参考其帮助文档。


图 6-41　GridView 运行效果

### 3. PathView

PathView 与 ListView、GridView 最大的不同之处在于它是在一个路径(Path)上排列条目的。再进一步讲解之前，先来看一个例子。(项目源码路径：src\6\6-57\myPathView)

```
import QtQuick 1.0

Rectangle {
  width: 400; height: 240
  color: "white"

  ListModel {
    id: appModel
    ListElement { name: "Music"; icon: "AudioPlayer_48.png" }
    ListElement { name: "Movies"; icon: "VideoPlayer_48.png" }
    ListElement { name: "Camera"; icon: "Camera_48.png" }
    ListElement { name: "Calendar"; icon: "DateBook_48.png" }
    ListElement { name: "Messaging"; icon: "EMail_48.png" }
    ListElement { name: "Todo List"; icon: "TodoList_48.png" }
    ListElement { name: "Contacts"; icon: "AddressBook_48.png" }
  }

  Component {
    id: appDelegate
    Item {
      width: 100; height: 100
      scale: PathView.iconScale
      Image {
        id: myIcon
        y: 20; anchors.horizontalCenter: parent.horizontalCenter
        source: icon
        smooth: true
      }
```

```
        Text {
          anchors { top: myIcon.bottom;
            horizontalCenter: parent.horizontalCenter }
          text: name
          smooth: true
        }
        MouseArea {
          anchors.fill: parent
          onClicked: view.currentIndex = index
        }
      }
    }

    Component {
      id: appHighlight
      Rectangle { width: 80; height: 80; color: "lightsteelblue" }
    }

    PathView {
      id: view
      anchors.fill: parent
      highlight: appHighlight
      preferredHighlightBegin: 0.5
      preferredHighlightEnd: 0.5
      focus: true
      model: appModel
      delegate: appDelegate
      path: Path {
        startX: 10
        startY: 50
        PathAttribute { name: "iconScale"; value: 0.5 }
        PathQuad { x: 200; y: 150; controlX: 50; controlY: 200 }
        PathAttribute { name: "iconScale"; value: 1.0 }
        PathQuad { x: 390; y: 50; controlX: 350; controlY: 200 }
        PathAttribute { name: "iconScale"; value: 0.5 }
      }
    }
  }
```

代码的运行效果如图 6-42 所示。现在先来看一下路径 Path 的概念。一个 Path 由一个或多个路径段组成，可用的路径包括 PathLine、PathQuad 和 PathCubic。其中，PathLine 定义了一条直线，可以以一条直线到达指定的位置。例如下面的代码中从(0，100)点做直线到(200，100)点：

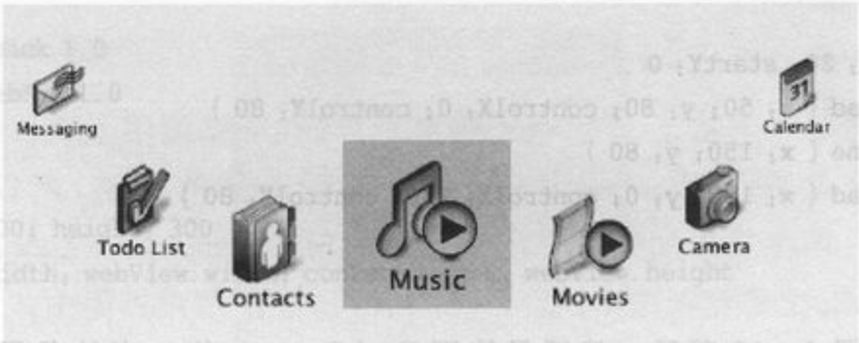



图 6-42　PathView 运行效果

```
Path {
  startX: 0; startY: 100
  PathLine { x: 200; y: 100 }
}
```

而 PathQuad 定义了包含一个控制点的二次贝赛尔曲线。例如：

```
Path {
  startX: 0; startY: 0
  PathQuad { x: 200; y: 0; controlX: 100; controlY: 150 }
}
```

效果如图 6-43 所示。而 PathCubic 定义了一个具有两个控制点的三次贝塞尔曲线。例如：

```
Path {
  startX: 20; startY: 0
  PathCubic {
    x: 180; y: 0
    control1X: -10; control1Y: 90
    control2X: 210; control2Y: 90
  }
}
```

其效果如图 6-44 所示。

图 6-43　二次贝塞尔曲线示意图

图 6-44　三次贝塞尔曲线示意图

Path 中条目之间的空隙使用 PathPercent 对象来调节。下面通过一个对比来看一下 PathPercent 的作用。首先是正常的条目分布：

```
PathView {
  ...
```

```
    Path {
        startX: 20; startY: 0
        PathQuad { x: 50; y: 80; controlX: 0; controlY: 80 }
        PathLine { x: 150; y: 80 }
        PathQuad { x: 180; y: 0; controlX: 200; controlY: 80 }
    }
}
```

其效果如图 6-45 所示。然后是使用 PathPercent 将一半的条目分布在中间的 PathLine 上：

```
PathView {
    ...
    Path {
        startX: 20; startY: 0
        PathQuad { x: 50; y: 80; controlX: 0; controlY: 80 }
        PathPercent { value: 0.25 }
        PathLine { x: 150; y: 80 }
        PathPercent { value: 0.75 }
        PathQuad { x: 180; y: 0; controlX: 200; controlY: 80 }
        PathPercent { value: 1 }
    }
}
```

其效果如图 6-46 所示。

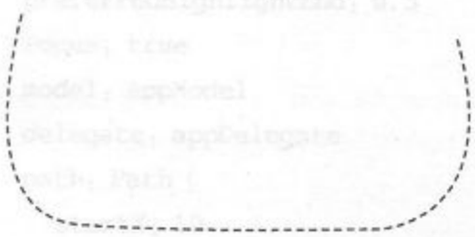


图 6-45　没有使用 PathPercent 的效果

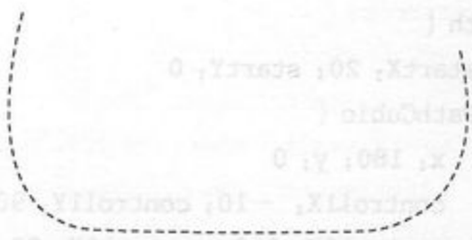


图 6-46　使用 PathPercent 的效果

Path 中还可以使用 PathAttribute，该对象允许在路径上的多个点指定由名称和值组成的属性。这些属性可以作为附加属性在委托中使用。在路径上其他没有指定属性的点会进行线性插值。例如在前面的例子中，指定了一个 iconScale 属性，分别在路径的两端和中间部位指定了其缩放值，中间显示的图片为正常大小，而两端的图片显示一半的大小。在端点和中间点之间的点会进行线性插值，也就是其缩放值在 0.5 和 1.0 进行线性变化。

**4. WebView**

WebView 主要用来对网页内容进行渲染，只需为其指定一个 URL 即可。在使用该元素时需要先导入 QtWebKit 模块。下面来看一个简单的例子。（项目源码路径：src\6\6-58\myWebView）

```
import QtQuick 1.0
import QtWebKit 1.0

Flickable {
    width: 400; height: 300
    contentWidth: webView.width; contentHeight: webView.height

    WebView {
        id: webView
        url: "http://www.baidu.com"
    }
}
```

因为 WebView 默认是不可以弹动的，所以要将其放入一个 Flickable 项目中。这里只需要指定网页的 URL 即可。这样就可以像一个浏览器一样来显示网页内容，而且还可以进行页面的拖动。对于更多的相关内容，可以查看 WebView 的帮助文档，也可以参考一下 Qt 提供的 WebView example 示例程序和 Web Browser 演示程序。

## 6.8　QML 和 C++混合编程

### 6.8.1　Qt 声明式用户界面运行环境

QML 文件通过 QML 运行环境进行加载和执行。这包括声明式用户界面引擎和内建的 QML 元素与插件模块，它也允许对第三方 QML 元素和模块的访问。使用 QML 的应用程序需要调用 QML 运行环境来执行 QML 文档，这可以通过创建 QDeclarativeView 或者 QDeclarativeEngine 来完成。另外，声明式用户界面包含了 Qt QML Viewer 工具，它可以用来加载. qml 文件。该工具可以用来开发和测试 QML 代码，而不需要编写 C++应用程序来加载 QML 运行环境。本章的开始已经介绍过该工具了，也可以在 Qt 帮助中查看 QML Viewer 关键字来查看更详细的介绍。对应本节的内容，可以在 Qt 帮助中参考 Qt Declarative UI Runtime 关键字。

要部署使用了 QML 的应用程序，必须在应用程序中调用 QML 运行环境。这可以通过编写一个 Qt C++应用程序，然后通过 QDeclarativeView 实例来加载 QML 文件，或者创建一个 QDeclarativeEngine 实例然后使用 QDeclarativeComponent 来加载 QML 文件。具体使用哪种方法依赖于已经存在的用户界面代码的特点。关于这些方法的详细介绍，可以在 Qt 帮助中查看 Integrating QML with existing Qt UI code 关键字。

### 1. 使用 QDeclarativeView 来部署

如果已经拥有一个基于 QWidget 的用户界面，可以使用 QDeclarativeView 将 QML 部件整合进来。QDeclarativeView 是 QWidget 的子类，所以可以像其他 QWidget 部件一样将其添加到用户界面中。使用 QDeclarativeView::setSource() 来加载一个 QML 文件到视图中，然后将该视图添加到用户界面中。下面来看一个例子。

(项目源码路径：src\6\6-59\myDeclarativeView)新建空的 Qt 项目，项目名称为 myDeclarativeView。然后向项目中添加新的 application.qml 文件，并将其内容更改如下：

```
import QtQuick 1.0
Rectangle { width: 100; height: 100; color: "red" }
```

然后向项目中添加新的 main.cpp 文件，并添加如下代码：

```
#include <QApplication>
#include <QDeclarativeView>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QDeclarativeView view;
    view.setSource(QUrl("../myDeclarativeView/application.qml"));
    view.show();

    return app.exec();
}
```

这样便创建了一个基于 QWidget 的视图来显示 application.qml 文件的内容。最后还要在项目文件 myDeclarativeView.pro 中添加如下代码：

```
QT += declarative
```

也就是说，必须添加 declarative 模块才可以在 Qt 程序中显示 QML 文件的内容。现在运行程序，效果如图 6-47 所示。使用这种方式的一个缺点是：与 QWidget 相比，QDeclarativeView 初始化很慢，而且会使用更多的内存。如果创建大量的 QDeclarativeView 对象会导致性能下降。如果发生了这种情况，一个比较好的方法是在 QML 中重写这些部件，然后在主 QML 部件中加载这些部件，而不要使用 QDeclarativeView。

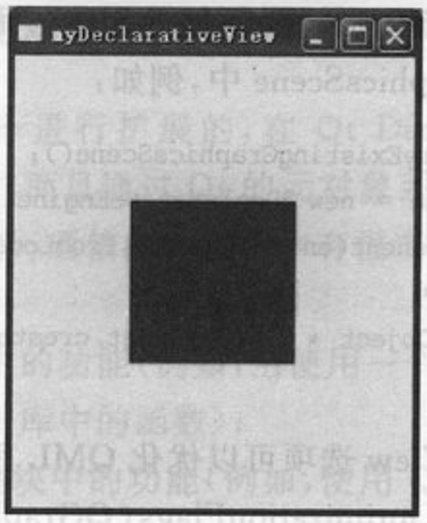



图 6-47　在 Qt 程序中显示 QML 文件内容

注意，与 QML 相比，QWidget 是为多种不同类型的用户界面设计的，所以将基于 QWidget 的应用程序和 QML 相连接并不总是一个好主意。如果用户界面是由少量复杂的静态元素组成，那么最好使用 QWidget 部件来实现；而如果用户界面由大量简单的和动态的元素组成，那么最好使用 QML 来实现。

**2. 直接创建一个 QDeclarativeEngine**

如果在 application.qml 中没有包含任何的图形组件，或者由于其他原因需要避免使用 QDeclarativeView，那么就可以直接创建 QDeclarativeEngine。在这种情况下，application.qml 会被作为一个 QDeclarativeComponent 实例进行加载，而不是显示在一个视图中。使用这种方式可以将主函数更改如下：

```
#include <QApplication>
#include <QDeclarativeEngine>
#include <QDeclarativeContext>
#include <QDeclarativeComponent>

int main(int argc, char *argv[])
{
  QApplication app(argc, argv);

  QDeclarativeEngine engine;
  QDeclarativeContext *objectContext =
                    new QDeclarativeContext(engine.rootContext());
  QDeclarativeComponent component(&engine, "application.qml");
  QObject *object = component.create(objectContext);

  //... 可以在必要的时候删除 object 和 objectContext

  return app.exec();
}
```

如果已经拥有了一个基于图形视图框架的用户界面，那么可以使用这种方式直接将QML部件整合到QGraphicsScene中，例如：

```
QGraphicsScene * scene = myExistingGraphicsScene();
QDeclarativeEngine * engine = new QDeclarativeEngine;
QDeclarativeComponent component(engine, QUrl::fromLocalFile("myqml.qml"));
QGraphicsObject * object =
  qobject_cast<QGraphicsObject *>(component.create());
scene->addItem(object);
```

使用下面的QGraphicsView选项可以优化QML用户界面的表现：

- QGraphicsView::setOptimizationFlags(QGraphicsView::DontSavePainterState)；
- QGraphicsView::setViewportUpdateMode(QGraphicsView::BoundingRectViewportUpdate)；
- QGraphicsScene::setItemIndexMethod(QGraphicsScene::NoIndex)。

还有一种方法是将现有的QGraphicsWidget对象暴露给QML，并且在QML中创建场景。关于这种方法的使用可以在Qt中参考Basic Graphics Layouts Example示例程序。

## 6.8.2 创建Qt Quick应用程序

前面已经讲解了如何在Qt应用程序中加载QML文件，那是通过创建空项目，然后手动添加文件和代码来实现的。其实在Qt Creator中，还可以直接创建Qt Quick应用程序，它会自动包含QML、C++代码以及QDeclarativeView，这种工程不仅可以在桌面环境下运行，还可以部署到手机等移动平台上。下面来看一个例子。

（项目源码路径：src\6\6-60\myQtQuick）新建Qt Quick应用程序，项目名称为myQtQuick。在应用程序选项中，可以在“自适应行为”中将屏幕横向或者纵向锁定，还可以指定应用程序图标等，这里保持默认。然后在QML源文件页面选择“生成一个main.qml文件”。创建完成后可以看到自动生成了很多文件，只需要关注main.cpp文件和main.qml文件即可。在main.qml文件中已经自动生成了Hello World的代码，现在运行程序，效果如图6-48所示。

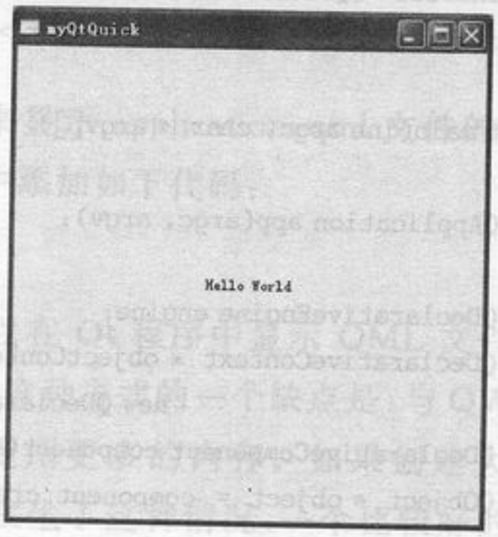



图6-48 Qt Quick应用程序程序运行效果

### 6.8.3　在 C++ 应用程序中使用 QML

QML 是很容易从 C++ 进行扩展的，在 Qt Declarative 模块中的类允许从 C++ 加载和操纵 QML 组件，而且通过 Qt 的元对象系统，QML 和 C++ 对象是可以很容易通过 Qt 信号和槽进行通信的。编程中有很多原因需要进行 QML 和C++ 的混合编程，例如：

- 使用定义在 C++ 源中的功能(例如，当使用一个 C++ 基于 Qt 的数据模型，或者调用第三方 C++ 库中的函数)；
- 访问 Qt Declarative 模块中的功能(例如，使用 QDeclarativeImageProvider 来动态创建图像)；
- 编写自己的 QML 元素。

要使用 Qt Declarative 模块，就必须先包含和链接合适的模块，关于如何使用该模块来创建一个基本的 C++ 应用程序已经在 6.8.1 小节讲到了。对应本节的内容，可以在 Qt 帮助文档中查看 Using QML in C++ Applications 关键字。

**1. 核心模块类**

Qt Declarative 模块提供了一组 C++ 接口来实现从 C++ 扩展 QML 应用程序，将 QML 嵌入到 C++ 应用程序中。在 Qt Declarative 模块中有几个核心类提供了完成该任务的基本功能。这些类是：

- QDeclarativeEngine：一个 QML 引擎提供了执行 QML 代码的环境，每一个应用程序至少需要一个引擎实例；
- QDeclarativeComponent：一个组件封装了一个 QML 文件；
- QDeclarativeContext：上下文允许应用程序将数据暴露给由引擎创建的 QML 组件。

QDeclarativeEngine 允许将全局设置应用到它的所有 QML 组件实例上，例如，QNetworkAccessManager 用于网络通信，一个文件路径用作永久性存储等。而 QDeclarativeComponent 用来加载 QML 文件，每一个 QDeclarativeComponent 实例表示一个独立的文件。一个 QDeclarativeComponent 可以从一个 URL、一个 QML 文件的路径或者原始的 QML 代码创建，可以通过 QDeclarativeComponent::create()函数进行实例化，例如：

```
QDeclarativeEngine engine;
QDeclarativeComponent component(&engine, QUrl::fromLocalFile("MyRectangle.qml"));
QObject * rectangleInstance = component.create();
//...
delete rectangleInstance;
```

QML 文件也可以使用 QDeclarativeView 加载，这个类提供了一个方便的基于

QWidget的视图，它将QML组件嵌入到一个基于QGraphicsView的应用程序。

**2. 混合使用QML和C++的方法**

这里提供了几种方法来通过C++扩展QML应用程序，例如：

- 加载一个QML组件，然后从C++对其进行操作；
- 直接将一个C++对象及其属性嵌入到QML组件；
- 定义一个新的QML元素（通过基于QObject的C++类）并在QML代码中创建它们。

下面分别介绍这几种方法，当然这些方法不具有排他性，也就是说可以在应用程序中同时使用多种方法。

**3. 从C++加载一个QML组件**

一个QML文件可以使用QDeclarativeComponent或者QDeclarativeView加载。QDeclarativeComponent将一个QML组件加载为一个C++对象；QDeclarativeView直接将QML组件加载到一个QGraphicsView中。例如，假设有一个MyItem.qml文件，其内容如下：

```
import QtQuick 1.0

Item {
  width: 100; height: 100
}
```

如果使用QDeclarativeComponent，这时需要调用QDeclarativeComponent::create()来创建该组件的一个新的实例：

```
QDeclarativeEngine engine;
QDeclarativeComponent component(&engine, QUrl::fromLocalFile("MyItem.qml"));
QObject *object = component.create();
...
delete object;
```

而如果使用QDeclarativeView，那么它会自动创建该组件的实例，后面可以通过函数QDeclarativeView::rootObject()来获取该实例：

```
QDeclarativeView view;
view.setSource(QUrl::fromLocalFile("MyItem.qml"));
view.show();
QObject *object = view.rootObject
```

这里的object是新创建的MyItem.qml组件的实例，可以使用QObject::setProperty()或者QDeclarativeProperty来修改该项目的属性：

```
object->setProperty("width", 500);
QDeclarativeProperty(object, "width").write(500);
```

另外，也可以先将该对象转换为真实的类型，然后调用编译时安全的函数。在这里 MyItem.qml 的基对象是一个 Item，它由 QDeclarativeItem 类进行定义：

```
QDeclarativeItem * item = qobject_cast<QDeclarativeItem *>(object);
item->setWidth(500);
```

QML 组件本质上是一个包含孩子的对象树，而孩子对象可以拥有兄弟和它自己的孩子。可以使用 QObject::objectName 属性和 QObject::findChild()函数来定位 QML 组件的子对象。例如，在 MyItem.qml 中的根项目有一个 Rectangle 子项目：

```
import QtQuick 1.0

Item {
  width: 100; height: 100
  Rectangle {
    anchors.fill: parent
    objectName: "rect"
  }
}
```

这里可以像这样来定位孩子：

```
QObject * rect = object->findChild<QObject *>("rect");
if (rect)
  rect->setProperty("color", "red");
```

如果 objectName 用在一个 ListView 或 Repeater 的委托中或者其他一些使用委托创建多个实例的元素中，那么会有多个孩子使用相同的 objectName。在这种情况下，可以使用 QObject::findChildren()来查找所有匹配 objectName 的孩子。

**4. 在 QML 组件中嵌入 C++对象**

当加载一个 QML 场景到 C++应用程序中时，直接嵌入 C++数据到 QML 对象中是很有用的。这个可以通过 QDeclarativeContext 来实现，它可以将数据暴露给 QML 组件的上下文，允许将数据从 C++注入到 QML 中。例如，这里有一个 QML 项目引用了一个 currentDateTime 值，而该值并没有在当前的作用域：

```
// MyItem.qml
import QtQuick 1.0

Text { text: currentDateTime }
```

这里的 currentDateTime 值可以直接在加载了该 QML 组件的 C++应用程序中设置，这需要使用 QDeclarativeContext::setContextProperty()：

```
QDeclarativeView view;
```

```
view.rootContext()->setContextProperty("currentDateTime",
                                        QDateTime::currentDateTime());
view.setSource(QUrl::fromLocalFile("MyItem.qml"));
view.show();
```

上下文属性可以使用 QVariant 或者 QObject * 值，这也就意味着自定义的C++对象也可以使用这种方式进行注入，而且这些对象可以直接在 QML 中进行读取和修改。下面来看一个例子。(项目源码路径：src\6\6-61\myDeclarativeContext)新建空的 Qt 项目，项目名称为 myDeclarativeContext。完成后在项目文件 myDeclarativeContext.pro 中添加如下代码：

```
QT += declarative
```

完成后保存该文件。然后再往项目中添加新的 applicationdata.h 文件，更改其内容如下：

```
#ifndef APPLICATIONDATA_H
#define APPLICATIONDATA_H

#include <QObject>
#include <QDateTime>

class ApplicationData : public QObject
{
  Q_OBJECT
public:
  Q_INVOKABLE QDateTime getCurrentDateTime() const {
    return QDateTime::currentDateTime();
  }
};

#endif // APPLICATIONDATA_H
```

需要说明，如果要在 QML 中调用一个C++函数，那么这个函数必须是一个 Qt 槽，或者该函数使用 Q_INVOKABLE 宏进行标记，这里就是使用了该宏对函数进行了标记。

下面向项目中添加新的 main.cpp 文件，添加如下代码：

```
#include <QApplication>
#include <QDeclarativeView>
#include <QDeclarativeContext>
#include "applicationdata.h"

int main(int argc, char *argv[])
{
```

```
    QApplication app(argc, argv);

    QDeclarativeView view;

    ApplicationData data;
    view.rootContext()->setContextProperty("applicationData", &data);

    view.setSource(QUrl::fromLocalFile("../myDeclarativeContext/MyItem.qml"));
    view.show();

    return app.exec();
}
```

这里使用 QDeclarativeContext::setContextProperty()设置了 applicationData 对象，可以在 QML 文件中直接访问它。最后向项目中添加新的 MyItem.qml 文件，更改其内容如下：

```
import QtQuick 1.0
Text { text: applicationData.getCurrentDateTime() }
```

现在运行程序，可以看到已经显示出时间信息了。这里就是在 QML 代码中直接使用了对象实例来调用一个函数。在这个例子中是在 QML 中调用了 C++中的函数，那么怎样在C++代码中调用 QML 组件中的函数呢？这个可以使用 QMetaObject::invokeMethod()函数来实现。例如，在 MyItem.qml 中有一个函数：

```
import QtQuick 1.0

Item {
    function myQmlFunction(msg) {
        console.log("Got message:", msg)
        return "some return value"
    }
}
```

那么在 C++应用程序中可以这样来使用该 QML 函数：

```
// main.cpp
QDeclarativeEngine engine;
QDeclarativeComponent component(&engine, "MyItem.qml");
QObject *object = component.create();
QVariant returnedValue;
QVariant msg = "Hello from C++";
QMetaObject::invokeMethod(object, "myQmlFunction",
    Q_RETURN_ARG(QVariant, returnedValue),
    Q_ARG(QVariant, msg));
qDebug() << "QML function returned:" << returnedValue.toString();
```

```
delete object;
```

注意，在QMetaObject::invokeMethod()函数的Q_RETURN_ARG()和Q_ARG()参数必须指定为QVariant类型，分别用于接收返回值和传递函数参数。

如果QML需要从上下文属性中接收一个信号，可以使用Connections元素进行关联。例如，如果ApplicationData拥有一个叫dataChanged()的信号，那么可以在Connections对象中使用一个onDataChanged处理器来关联该信号：

```
Text {
    text: applicationData.getCurrentDateTime()
    Connections {
        target: applicationData
        onDataChanged: console.log("The application data changed!")
    }
}
```

而对于QML中的信号在C++中是自动可用的，可以像关联普通的Qt C++信号一样来对其进行关联。例如，在MyItem.qml中有一个qmlSignal()信号：

```
import QtQuick 1.0

Item {
    id: item
    width: 100; height: 100

    signal qmlSignal(string msg)

    MouseArea {
        anchors.fill: parent
        onClicked: item.qmlSignal("Hello from QML")
    }
}
```

假设在C++代码中有一个MyClass类，它有一个cppSlot()槽，那么可以这样进行关联：

```
int main(int argc, char * argv[]) {
    QApplication app(argc, argv);

    QDeclarativeView view(QUrl::fromLocalFile("MyItem.qml"));
    QObject * item = view.rootObject();

    MyClass myClass;
    QObject::connect(item, SIGNAL(qmlSignal(QString)),
                     &myClass, SLOT(cppSlot(QString)));
```

```
    view.show();
    return app.exec();
}
```

**5. 定义一个新的 QML 元素**

新的 QML 元素可以在 QML 中进行定义，它们也可以使用 C++类来定义。事实上，很多核心 QML 元素都是通过 C++类实现的。当使用这些元素中的一个来创建一个 QML 对象时，也就是创建了一个基于 QObject 的 C++类的实例并且设置了属性。例如，下面的 ImageViewer 类有一个 image 路径属性：

```
#include <QtCore>
#include <QtDeclarative>

class ImageViewer : public QDeclarativeItem
{
  Q_OBJECT
  Q_PROPERTY(QUrl image READ image WRITE setImage NOTIFY imageChanged)

public:
  void setImage(const QUrl &url);
  QUrl image() const;

signals:
  void imageChanged();
};
```

除了该类继承自 QDeclarativeItem 外，它与在 QML 外部存在的普通的类没有什么区别。然而，一旦使用 qmlRegisterType()使其在 QML 引擎进行注册：

```
qmlRegisterType<ImageViewer>("MyLibrary", 1, 0, "ImageViewer");
```

这时，任何在 C++应用程序或者插件中加载的 QML 代码都可以创建一个 ImageViewer 对象：

```
import MyLibrary 1.0

ImageViewer { image: "smile.png" }
```

注意，自定义的C++类型不一定非要继承自 QDeclarativeItem，只有在其是一个可显示的项目时才是必须的。如果该项目是不可以显示的，那么它可以继承自 QObject。更多关于定义新的 QML 元素的内容，可以在 Qt 帮助中查看 Writing QML extensions with C++关键字，这里提供了一个详细的教程；也可以参考 Extending QML in C++关键字对应的帮助文档。

### 6.8.4 QML中的全局对象

QML中的全局对象包含了所有JavaScript全局对象的属性，以及Qt对象、XMLHttpRequest对象、离线存储接口和记录输出函数(console.log()和console.debug())。对于XMLHttpRequest和离线存储接口的介绍，可以在Qt帮助中参考本节对应的QML Global Object文档。下面来看一下Qt对象。

QML全局Qt对象提供了Qt中非常有用的枚举变量和函数。Qt对象不是一个QML元素，它不能被实例化。如果要使用它，可以直接调用全局Qt对象的函数，例如：

```
import QtQuick 1.0

Text {
  color: Qt.rgba(1, 0, 0, 1)
  text: Qt.md5("hello, world")
}
```

可以在Qt帮助中查看QML Qt Element关键字来查看全局Qt对象中所有的函数。

**1. 枚举变量**

Qt对象包含了在Qt的元对象系统中声明的枚举变量。例如，可以使用Qt.LeftButton来访问Qt::MouseButton枚举变量中的LeftButton成员。

**2. 类型**

Qt对象也包含了辅助函数用于创建指定数据类型的对象。这主要应用在当设置项目的属性而该属性中包含了下面的类型时：

- color：使用Qt.rgba()、Qt.hsla()、Qt.darker()、Qt.lighter()、Qt.tint()；
- rect：使用Qt.rect()；
- point：使用Qt.point()；
- size：使用Qt.size()；
- vector3d：使用Qt.vector3d()。

**3. 日期/时间格式化**

Qt对象中包含了一些函数用来格式化QDateTime、QDate和QTime的值。

- string Qt.formatDateTime(datetime date, variant format)
- string Qt.formatDate(datetime date, variant format)
- string Qt.formatTime(datetime date, variant forma

这里的格式使用dd.MM.yyyy hh:mm:ss.zzz等形式来表示。具体的内容可以在QML Qt Element关键字对应的文档中进行查看。

**4. 动态对象创建**

全局对象中提供了函数来允许从文件或字符串动态创建 QML 项目：

- object Qt. createComponent(url)
- object Qt. createQmlObject(string qml, object parent, string filepath)

## 6.9　使用 Qt Quick 设计器

前面都是使用纯代码的方式来编写 QML 程序的。除此之外，Qt Creator 中还提供了一个 Qt Quick 设计器，提供了可视化的方法来编辑 QML 文件。打开一个项目后，可以双击 QML 文件在代码编辑器中打开它，这时选择设计模式便可以在可视化编辑器中编辑该文件。设计器的界面如图 6－49 所示。

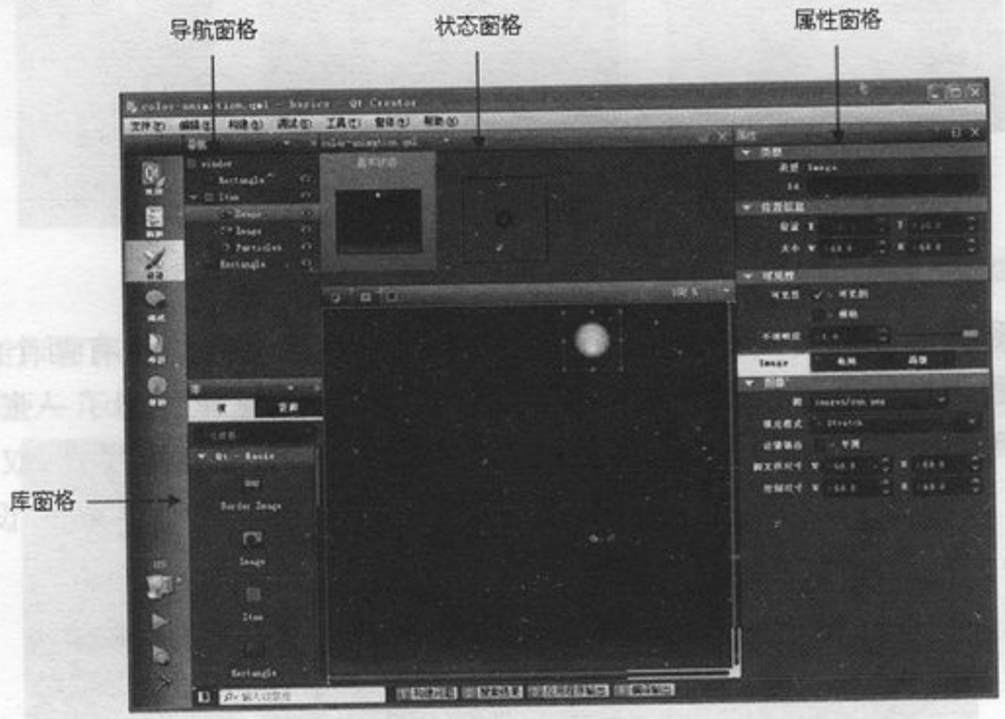



图 6－49　Qt Quick 设计器界面

可以看到 Qt Quick 设计器由中央的画布和几个窗格组成。可以使用这些可视化的编辑器窗格来管理项目：

- 导航窗格：使用树结构来显示当前 QML 文件中的 QML 元素；
- 库窗格：用来显示可以设计应用程序的构建块：预定义的 QML 元素、自定义的 QML 组件和图片等其他资源；
- 属性窗格：用来组织选中的 QML 元素或者 QML 组件的属性，可以在这里修改属性；
- 状态窗格：显示组件的不同的状态。

下面来看一个使用 Qt Quick 设计器编辑 QML 文件的例子，这个例子在一个页面上显示了一个 Qt 图标和 3 个矩形，单击一个矩形时 Qt 图标便移动到该矩形中。

(项目源码路径:src\6\6-62\Transitions)

**1. 创建项目**

在Qt Creator中创建Qt Quick项目,模板选择Qt Quick UI,项目名称为Transitions。这时可以看到Qt Creator已经生成了默认的QML文件供我们修改从而创建应用程序的主视图,如图6-50所示。

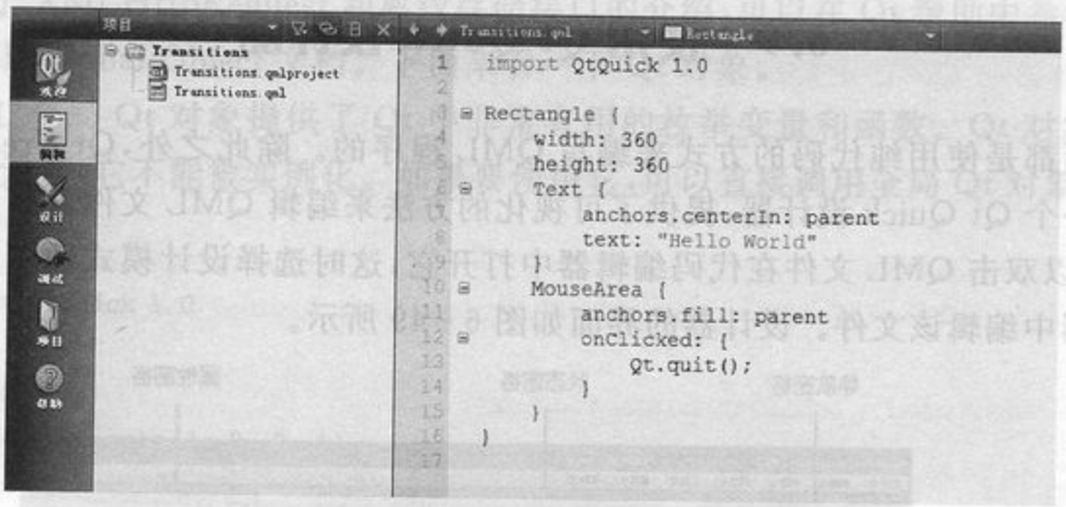



图6-50 自动生成的QML文件

**2. 创建主视图**

应用程序的主视图在屏幕的左上角显示了一个Qt图标,还有两个空的矩形。要在应用程序中使用图片,需要将其复制到项目目录中,这里复制了一张qt-logo.png图片到项目目录中,图片会出现在库窗格的资源中。在编辑模式,双击Transitions.qml文件在代码编辑器中打开它,然后单击设计模式在Qt Quick设计器中打开该文件,如图6-51所示。

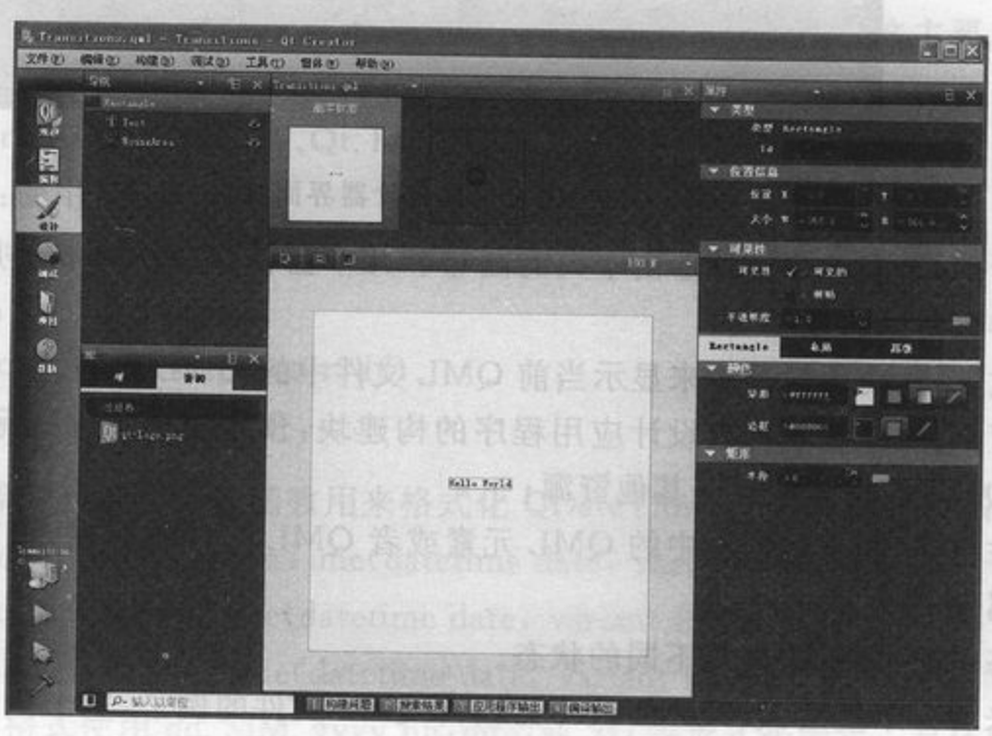



图6-51 在设计模式打开文件

然后在导航窗格中选择 Text，并按下键盘的 Delete 键删除该元素。然后选中 Rectangle 来编辑其属性：在 Id 域中输入 page，这样便可以在其他地方引用该部件了。在颜色选项卡中，将矩形域的颜色设置为 #343434，如图 6－52 所示。

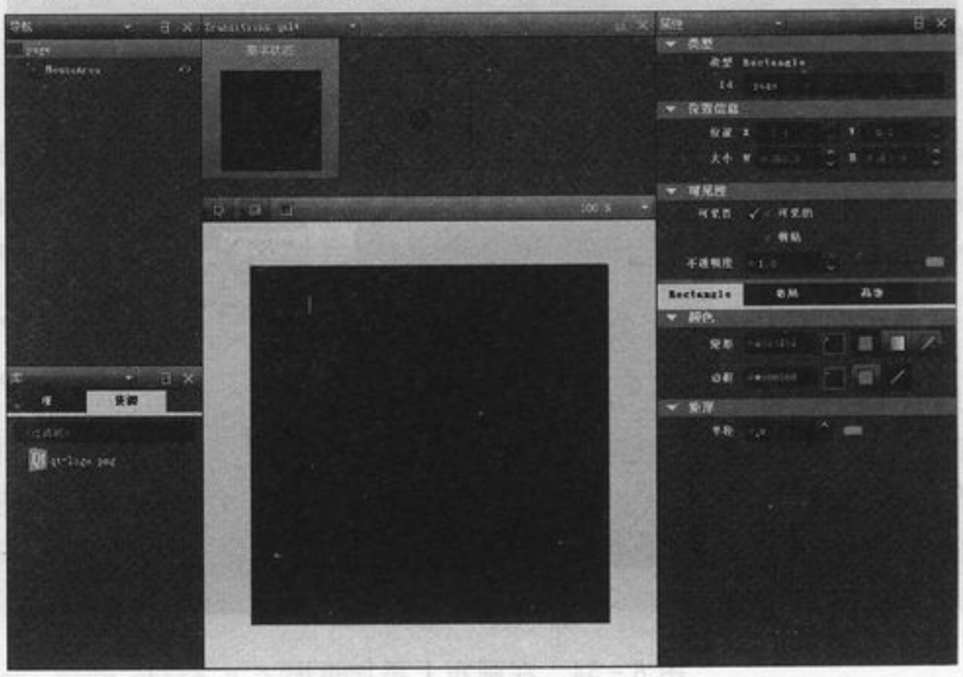


图 6－52　修改主视图界面

然后在库窗格的资源选项卡中选择 qt－logo. png 图片，并将其拖入画布中。在其 Id 域中输入 icon；在位置域中，设置 X 为 10，Y 为 20，如图 6－53 所示。

图 6－53　在画布上添加图片

在库窗格的 Item 选项卡中选择 Rectangle，将其拖拽到画布上并修改其属性：在 Id 域，输入 topLeftRect；在大小域中将 W 和 H 都设置为 64，让矩形的大小匹配图片的大小；在颜色选项卡的矩形域中单击按钮来使矩形变为透明，在边框域中将其设置为 #808080；在矩形选项卡的边框域中将边框宽度设置为 1。注意，如果在设置

了边框颜色后边框域没有出现在矩形选项卡中，可以尝试按下按钮将边框颜色设置为纯色；然后在半径域中，选择 6 来为矩形创建一个圆角，如图 6-54 所示。

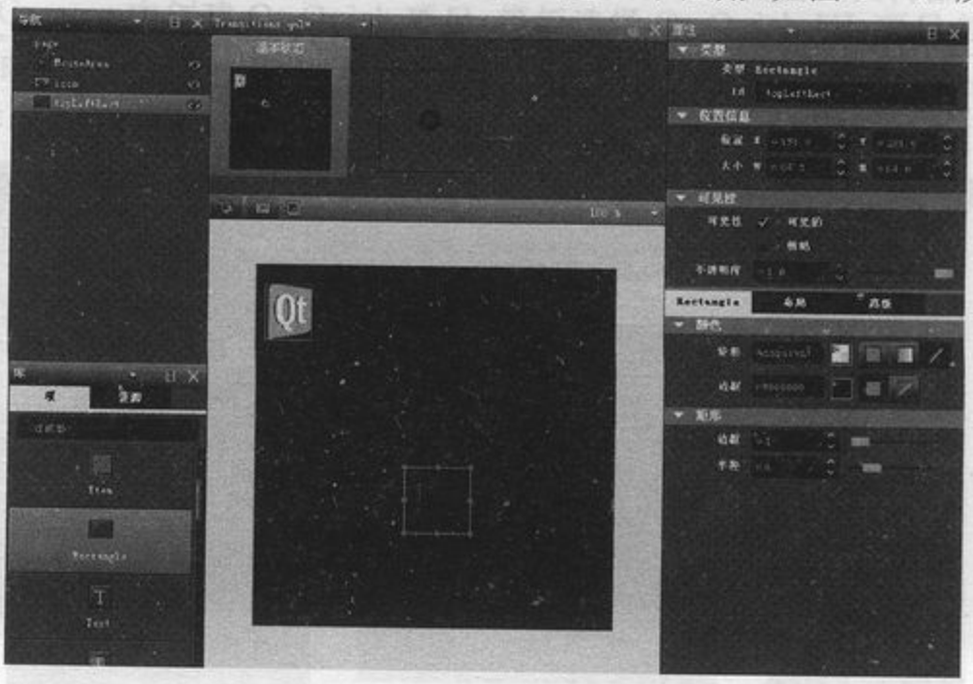


图 6-54　在画布上添加矩形

单击属性窗格中的布局，然后单击顶部和左部锚按钮来将矩形固定在页面的左上角。在页面空白域中将顶部锚设置为 20，左部锚设置为 10。然后在导航窗格中将 MouseArea 元素从 page 拖拽到 topLeftRect，使得它只应用在这个矩形上而不是整个页面，如图 6-55 所示。

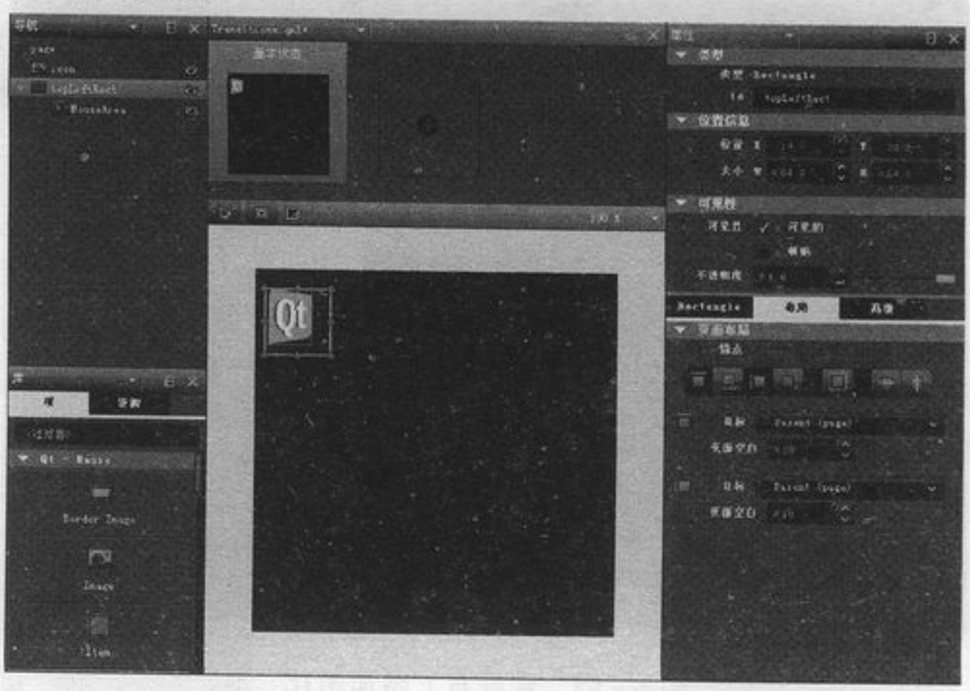


图 6-55　修改矩形的布局

下面来修改 MouseArea 的属性。在其布局中，单击按钮来使 MouseArea 填充整个矩形，然后转到编辑模式，可以看到这里的代码已经根据设计器中的改变而更改了。转到 MouseArea 代码部分，更改其按下时的表达式：

```
MouseArea {
    anchors.fill: parent
    onClicked: page.state = ''
}
```

这个表达式设置状态为默认状态，这样可以将图片返回到其初始位置。

下面回到设计模式，在导航窗格中，复制 topLeftRect(使用 Ctrl+C)并在画布上粘贴两次(使用 Ctrl+V)。Qt Creator 会自动重命名新的实例为 topLeftRect1 和 topLeftRect2。选择 topLeftRect1 并编辑其属性：在 Id 域输入 middleRightRect；在布局中，选择右部和垂直居中锚按钮，在页面空白域中，为右部锚选择 10，为垂直居中锚选择 0。然后到代码编辑器中设置其 MouseArea 按下时将状态更改为 State1：

```
onClicked: page.state = 'State1'
```

然后到设计模式中选择 topLeftRect2 并编辑其属性：在 Id 域中输入 bottomLeftRect；在布局中，选择底部和左部锚按钮，在页面空白域中，为底部锚选择 20，为左部锚选择 10。然后到代码编辑器中设置其 MouseArea 按下时将状态更改为 State2：

```
onClicked: page.state = 'State2'
```

现在设计模式的内容如图 6－56 所示。也可以先按下 Ctrl+S 保存更改，然后再按下 Ctrl+R 来运行应用程序。

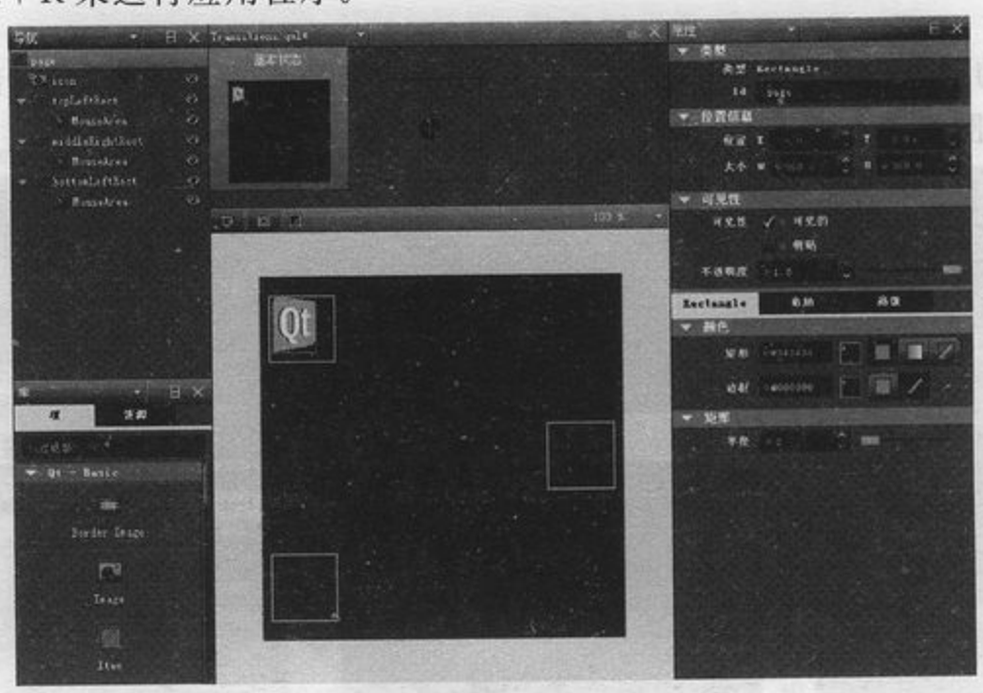


图 6－56　添加完矩形后的设计模式

### 3. 添加视图

QML 文件中已经创建了两个附加状态的指针：State1 和 State2，下面来创建这两个状态。首先在状态窗格中单击两次空槽来分别创建 State1 和 State2，默认生成的可能是中文，可以在其名称上双击再更改。完成后转到代码编辑器中绑定 Qt 图

标和矩形的位置，这样可以确保在窗口进行缩放时图标也可以显示在矩形中。修改的代码片段为：

```
states: [
  State {
    name: "State1"

    PropertyChanges {
      target: icon
      x: middleRightRect.x
      y: middleRightRect.y
    }
  },
  State {
    name: "State2"

    PropertyChanges {
      target: icon
      x: bottomLeftRect.x
      y: bottomLeftRect.y
    }
  }
]
```

现在的设计模式如图 6-57 所示。可以再次运行程序出现，已经可以单击矩形来移动 Qt 图标了。

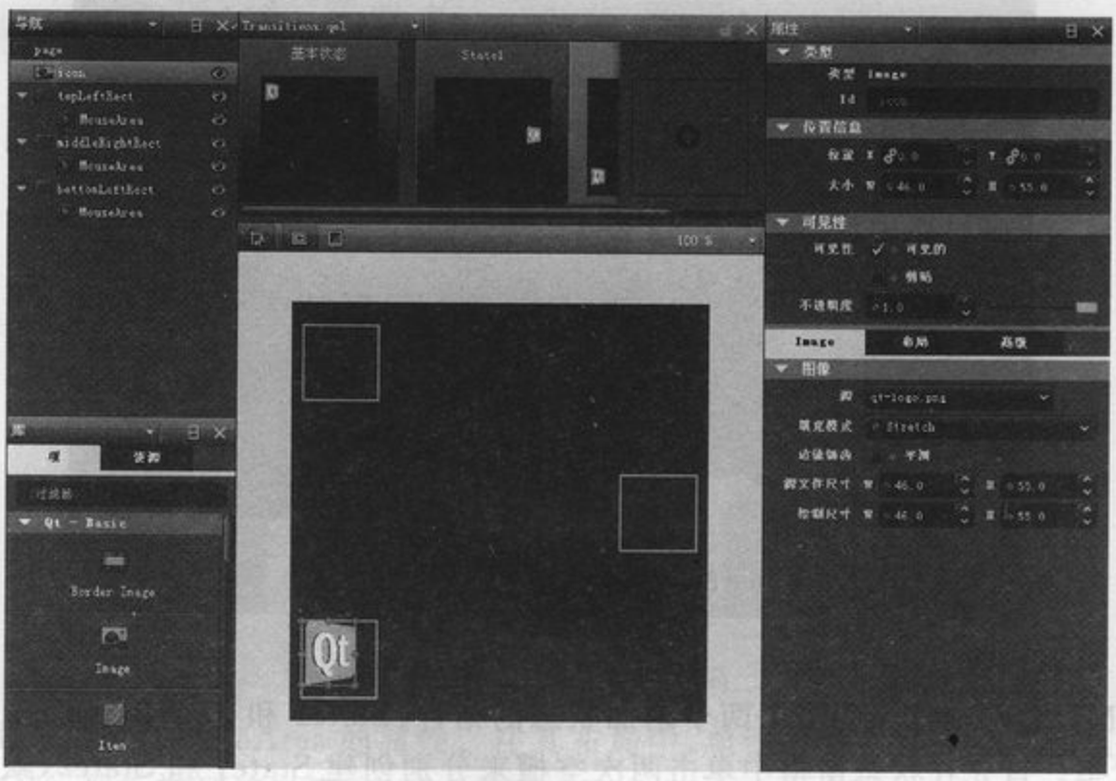



图 6-57　添加完状态后的设计模式

**4. 为视图添加动画**

可以添加切换来定义当 Qt 图标在不同的状态间移动时怎样改变属性，切换可以为 Qt 图标设置动画。在代码编辑器中，添加如下代码：

```
transitions: [
  Transition {
    from: "*"; to: "State1"
    NumberAnimation {
      properties: "x,y";
      duration: 1000
    }
  },

  Transition {
    from: "*"; to: "State2"
    NumberAnimation {
      properties: "x,y";
      easing.type: Easing.InOutQuad;
      duration: 2000
    }
  },

  Transition {
    NumberAnimation {
      properties: "x,y";
      duration: 200
    }
  }
]
```

其中，第一个 Transition 用来指定在向 State1 移动时，Qt 图标的 x 和 y 坐标进行线性变化，并且移动在 1 秒钟内完成；而第二个 Transition 用来指定向 State2 移动时，Qt 图标的 x 和 y 坐标的变化符合 InOutQuad 曲线，动画持续 2 秒；而第三个 Transition 用来指定其他的状态改变时，Qt 图标的 x 和 y 坐标进行线性变化，动画持续 200 毫秒。

这里可以使用动画元素的 Qt Quick 工具栏来改变缓和曲线类型。在 NumberAnimation 上单击就会出现图标，如图 6-58 所示。单击该图标，这时就会出现工具栏，如图 6-59 所示，然后选择动画为 Bounce，选择子类型为 Out，此时会自动添加相关代码。

```
transitions: [
    Transition {
        from: "*"; to: "State1"
        NumberAnimation {
            easing.type: Easing.OutBounce
            properties: "x,y";
            duration: 1000
        }
    },
```

图 6-58　设置弹出工具栏

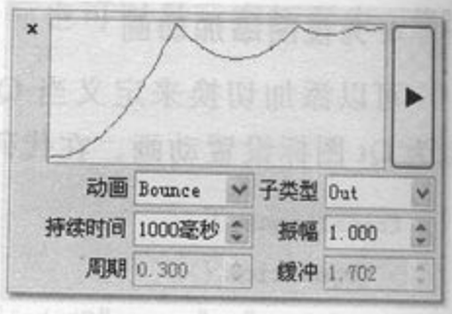



图 6-59　动画元素的工具栏

到这里，整个程序就设计完成了，现在可以运行程序查看效果。

## 6.10　小　结

本章详细介绍了 Qt Quick 各个方面的内容，主要讲解了 QML 语言的语法和基本的内建元素。因为篇幅的原因，我们并没有讲解大型的实例，读者可以在 Qt 帮助中查看 QML Examples and Demos 关键字对应的文档，这里列出了 Qt Quick 所有的示例程序和演示程序。

通过本章的介绍可以看到，QML 几乎涵盖了前面 Qt C++的所有内容，使用它创建一个动态的触摸式界面是非常容易的，而对于很多使用 Qt C++代码很难实现的功能，使用 QML 却很容易就可以实现，而且 QML 还可以很容易地使用 C++进行扩展。基于这些优点，Qt Quick 成为了 Qt 将来发展的核心内容。Qt Quick 作为一项全新的技术，现在还存在着很多不足的地方，不过 Qt 后面的版本一直在对其进行加强和完善。如果要学习 Qt 开发，特别是要进行移动应用程序开发，那么 Qt Quick 是一个很不错的选择。

# 参考文献

[1] 布兰切特,萨墨菲尔德. C++ GUI Qt4 编程[M]. 第 2 版. 闫锋欣,等,译. 北京:电子工业出版社,2008.

[2] 蔡志明,卢传富,李立夏,等. 精通 Qt4 编程[M]. 北京:电子工业出版社,2008.

[3] 成洁,卢紫毅. Linux 窗口程序设计——Qt4 精彩实例分析[M]. 北京:清华大学出版社,2008.

[4] Stanley B. Lippman Barbara E. Moo Josee LaJoie. C++ Primer 中文版[M]. 第 4 版. 李师贤,等,译. 北京:人民邮电出版社,2006.

[5] 王珊,萨师煊. 数据库系统概论[M]. 第 4 版. 北京:高等教育出版社,2006.

[6] 谢希仁. 计算机网络[M]. 第 5 版. 北京:电子工业出版社,2008.

策划编辑：董立娟
封面设计：runsign盛正设计·林健

Qt应用编程系列丛书

✓《Qt及Qt Quick开发实战精解》
《Qt Creator快速入门》

## 内容简介

本书主要讲解了5个Qt综合应用程序的开发过程和Qt Quick的相关内容。主要包括两部分：第一部分是多文档编辑器、方块游戏、音乐播放器、数据管理系统、局域网聊天工具这5个实用的Qt实例的详细讲解；第二部分是Qt Quick技术的全面介绍。

## 读者对象

本书的内容全面、实用，讲解通俗易懂，适合有一定Qt基础并且想学习Qt综合实例开发或者想学习Qt Quick技术的读者。对于没有Qt基础的读者，可以先学习《Qt Creator快速入门》一书。

## 共享资料

本书配套源码可以从www.yafeilinux.com下载，还可以到Qt爱好者社区（www.qter.org）的本系列丛书专版讨论交流。

## 作者简介

霍亚飞，网名yafeilinux，嵌入式软件工程师，热爱编程，热爱开源！在博客中发表了大量Qt、Linux教程和开源软件，被众多网友奉为经典！参与创建了www.yafeilinux.com和Qt爱好者社区(www.qter.org)，进行Qt及开源项目的推广和普及！

上架建议：程序设计/嵌入式系统

ISBN 978-7-5124-0781-7

9 787512 407817 >

定价：36.00元